



计 算 机 科 学 从 书

P Pearson

原书第5版

数据结构与抽象 Java语言描述

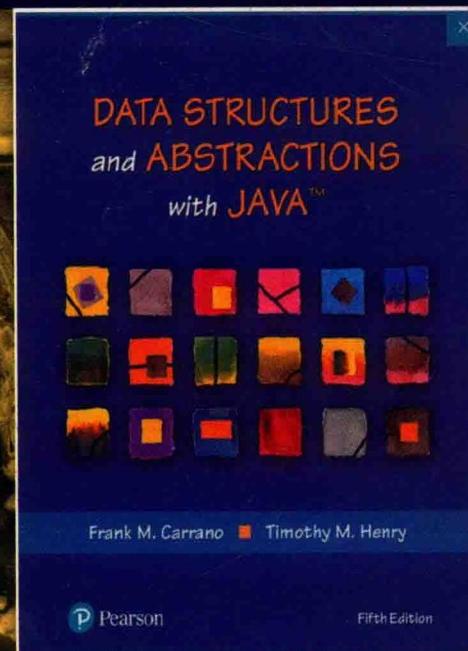
弗兰克·M. 卡拉诺 (Frank M. Carrano)

罗得岛大学

[美] 蒂莫西·M. 亨利 (Timothy M. Henry) 著 辛运伟 译

新英格兰理工学院

经典数据结构教材新版，以Java语言与数据结构两条知识主线贯穿始终



Data Structures and Abstractions with Java
Fifth Edition

计 算 机 科 学 从 书

原书第5版

数据结构与抽象 Java语言描述

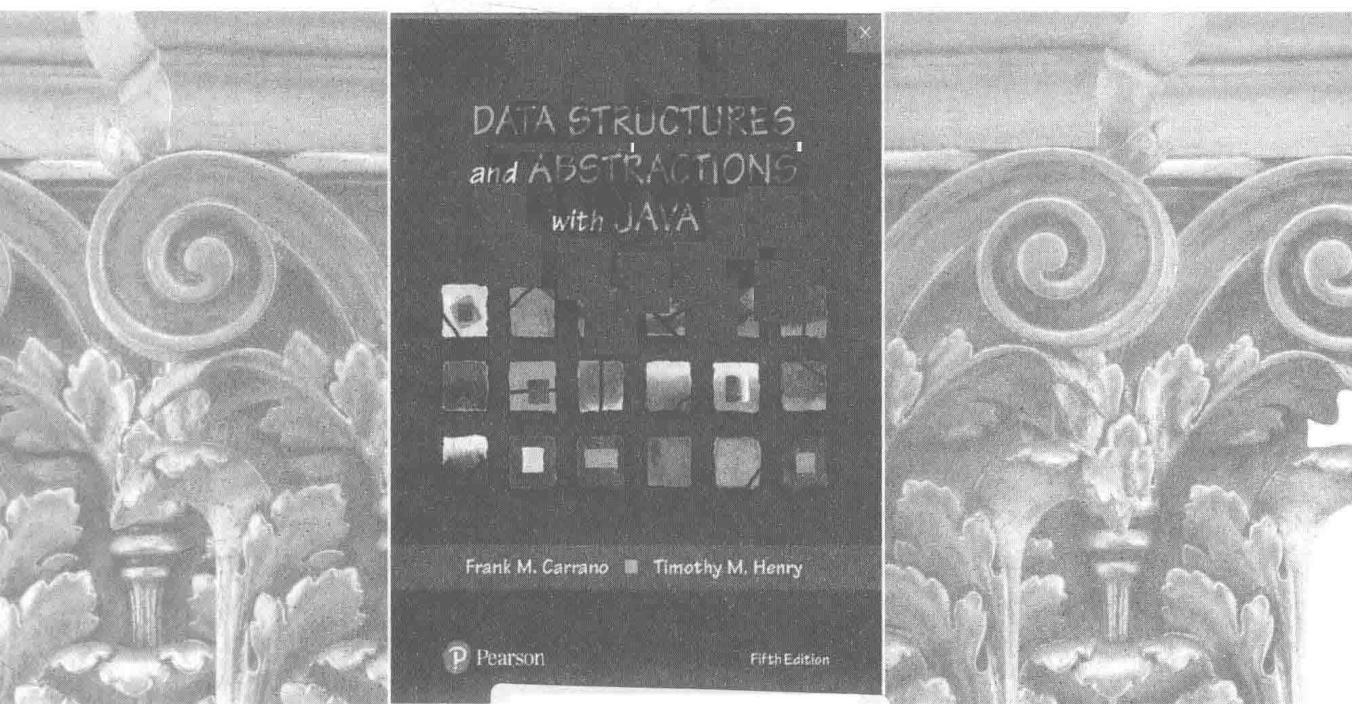
弗兰克·M. 卡拉诺 (Frank M. Carrano)

[美] 蒂莫西·M. 亨利 (Timothy M. Henry) 著 辛运伟 译

新英格兰理工学院

Data Structures and Abstractions with Java

Fifth Edition



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

数据结构与抽象：Java 语言描述（原书第 5 版）/（美）弗兰克·M. 卡拉诺（Frank M. Carrano），（美）蒂莫西·M. 亨利（Timothy M. Henry）著；辛运伟译。—北京：机械工业出版社，2019.8

（计算机科学丛书）

书名原文：Data Structures and Abstractions with Java, Fifth Edition

ISBN 978-7-111-63637-3

I. 数… II. ① 弗… ② 蒂… ③ 辛… III. ① 抽象数据结构 - 高等学校 - 教材 ② JAVA 语言 - 程序设计 - 高等学校 - 教材 IV. ① TP311.12 ② TP312.8

中国版本图书馆 CIP 数据核字（2019）第 200653 号

本书版权登记号：图字 01-2018-6313

Authorized translation from the English language edition, entitled *Data Structures and Abstractions with Java, Fifth Edition*, 978-0134831695 by Frank M. Carrano, Timothy M. Henry, published by Pearson Education, Inc., Copyright © 2019, 2015, 2012 and 2007 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by China Machine Press, Copyright © 2019.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

本书介绍数据结构及其 Java 实现，内容分为 30 章，涵盖的 ADT 有包、栈、队列等线性结构以及二叉树、查找树等层次结构和图结构，给出了对应于各 ADT 规范说明的 Java 类，分别基于数组方式和指针方式实现了类中的主要操作。书中还介绍了排序、查找相关算法，描述了字典、散列等概念和实现方式，介绍了评价算法效率的复杂度概念，以及使用迭代和递归方式实现算法的基本思路。

本书可作为相关专业数据结构课程的教材使用，也可供专业技术人员参考。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：朱秀英

责任校对：李秋荣

印 刷：北京瑞德印刷有限公司

版 次：2019 年 10 月第 1 版第 1 次印刷

开 本：185mm×260mm 1/16

印 张：47.75

书 号：ISBN 978-7-111-63637-3

定 价：139.00 元

客服电话：(010) 88361066 88379833 68326294

投稿热线：(010) 88379604

华章网站：www.hzbook.com

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本法律顾问：北京大成律师事务所 韩光 / 邹晓东

保留字

保留字也称为**关键字**。不能重新定义保留字，它们的含义不能改变。尤其是不能将保留字用作变量名、方法名或是类名。`null`都是字面值而不是保留字。但对我们而言，可以将它们看作保留字。

| | | | | |
|-----------|------------|-----------|--------------|------------|
| abstract | assert | boolean | break | byte |
| case | catch | char | class | const |
| continue | default | do | double | else |
| enum | extends | false | final | finally |
| float | for | goto | if | implements |
| import | instanceof | int | interface | long |
| native | new | null | package | private |
| protected | public | return | short | static |
| strictfp | super | switch | synchronized | this |
| throw | throws | transient | true | try |
| void | volatile | while | _ (下划线) | |

运算符优先级

下面的列表中，同一行的运算符有相同的优先级。表中越往下，优先级越低。当运算的次序不受括号控制时，较高优先级运算符的计算先于较低优先级运算符的计算。当运算符有相同优先级时，二元运算符按自左至右的次序进行，而一元运算符按自右至左的次序进行。

| | 高 |
|--|---|
| 一元运算符 +、-、++、--、!、~ | |
| 一元运算符 new 和 (类型) | |
| 二元运算符 *、/、% | |
| 二元运算符 +、- | |
| 二元(移位)运算符 <<、>>、>>> | |
| 二元运算符 <、>、<=、>= | |
| 二元运算符 ==、!= | |
| 二元运算符 & | |
| 二元运算符 ^ | |
| 二元运算符 | |
| 二元运算符 && | |
| 二元运算符 | |
| 三元(条件)运算符 ?: | |
| 赋值运算符 =、*=、/=、%=、+=、-=、<<=、>>=、>>>=、&=、^=、 = | 低 |

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson、McGraw-Hill、Elsevier、MIT、John Wiley & Sons、Cengage 等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出 Andrew S. Tanenbaum、Bjarne Stroustrup、Brian W. Kernighan、Dennis Ritchie、Jim Gray、Alfred V. Aho、John E. Hopcroft、Jeffrey D. Ullman、Abraham Silberschatz、William Stallings、Donald E. Knuth、John L. Hennessy、Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为本书的中译本作序。迄今，“计算机科学丛书”已经出版了近500个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们 的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章教育

华章科技图书出版中心

译者序

Data Structures and Abstractions with Java, Fifth Edition

时隔三年多，这本书又一次摆上案头。看着崭新的书，一方面敬佩原作者 Frank M. Carrano 和 Timothy M. Henry 严谨的治学态度和勤奋的工作精神，一方面也激励译者不能懈怠，要以更加认真的工作态度完成第 5 版的翻译工作。

一如既往，第 5 版也是个厚重的大部头。相比于第 4 版，主体内容没有太多的变化，仍然是 Java 语言与数据结构两条知识主线贯穿始终。本版的修订主要集中在三个方面。

第一方面，是章节次序做了调整，相关的主题集中介绍。这样，学习时内容连贯，也可以相互对比。比如，栈和队列都是受限的线性表，属于线性结构，通常情况下是依次介绍的。在第 4 版中，这两部分内容并不是连续的，其间还穿插介绍了递归和排序的内容。很明显，第 5 版的安排更加合理了。再比如，排序的方法比较多，按照效率来分，通常分为两大类，第 5 版中将排序方法也调整到了连续的章节中。

第二方面，是将递归的内容拆分为两章，先介绍基础知识，后介绍使用递归求解问题的方法。递归不是一个直观的概念，要想在程序中正确使用递归求解问题，深入理解概念是必不可少的环节。第 5 版加强了这些内容的介绍，帮助读者递归地思考问题，同时还介绍了递归程序的效率分析。

第三方面，是增加了侧重于游戏、电子商务及财务方面的练习和程序设计项目。学习了数据结构的知识之后，可以通过练习和程序设计项目来检验学习的效果。第 4 版中，在每章的最后都有数量不等的练习题和程序设计题目，在此基础上，第 5 版又增加了结合实际应用的题目。以这些题目作为切入点，可以逐步过渡到大的应用项目的开发。

第 5 版中，语言更加准确简捷，给出的图更加清晰明确。读者可以选择按序学习各章的内容，也可以先学习 ADT 的部分，再学习各 ADT 的实现部分。作者对章节的安排方便了读者的学习过程。作为教师，讲授的次序也可以自主安排。

为了让读者对全书内容有个大致的了解，下面简略地介绍一下全书的内容。

全书介绍了包、栈、队列、线性表、字典、树、堆及图等数据结构。每种数据结构都在连续的两三章中介绍。前一章专门介绍与每种结构相关的 ADT 的规范说明，包括数据属性及相关操作。使用具体实例说明这些操作的含义及操作的结果，帮助读者理解其定义。定义的接口也与规范说明放在同一章中，这一章的内容基本上不涉及实现的细节。接下来的一章或两章中，着重介绍对所讨论 ADT 的实现过程，并对每种实现进行效率分析。对有些重要且应用广泛的数据结构，还给出了问题求解实例。

树和图是非常重要的数据结构，树又衍生出二叉树、查找树等结构。它们的概念与实现，既来源于树，又有别于树，所以单独成章、重点介绍。图是本书介绍的最后一类结构，除了像其他 ADT 一样介绍了基本操作的实现之外，还着重介绍了图的几个应用，包括遍历、拓扑排序及最短路径问题。各类算法的实现，有很多是使用递归完成的。本书使用两章的篇幅介绍了递归的概念，及如何编写递归程序。

数据的存储方式可分为两大类：一类使用数组，包括向量；另一类使用指针，即链式实现。本书对各个 ADT 均给出基于数组及基于链的两种实现方式，并针对每种实现的适用性

做了介绍和对比。

除此之外，本书还介绍了排序、查找、散列等概念及相应的实现过程。它们是常用的方法，也是数据结构课程的重要组成部分。针对数据的不同存储方式，适用的排序和查找方法及其实现过程不尽相同，本书也一一做了介绍。

算法的效率是重要的衡量指标，作者介绍了算法分析的大 O 表示。书中实现的每个算法都使用这些技术进行了效率分析，特别介绍了递归方法的效率分析。

本书将 Java 语言的相关内容组织成插曲及附录，在插曲中介绍了泛型、异常、迭代器、可变对象及不可变对象、克隆等内容，在附录中介绍了 Java 的基本语法，以及编写 Java 程序时要注意的方面。同时还给出了词汇表，方便读者查询。附录中的内容属于 Java 语言的基本知识，插曲中的内容是更深入的部分。这些主题比较独立，熟悉这些内容的读者可以略过它们，也可以在涉及相关内容时作为参考而有选择地阅读。这样可以减轻读者的学习难度，也避免了顾此失彼。作者特别注重知识的衔接，在每章的最前面，列出学习本章时应该先学习哪些先修章节，这为读者画出了一个清晰的学习路线图。

译者在翻译过程中，尽可能地保持了原书的风格，包括排版格式。除了正文中的内容全部翻译外，也翻译了伪语言描述的算法中的英文语句，以帮助读者理解伪语言。

本书非常适合作为大学本科数据结构课程的教材使用。书中内容图文并茂，讲解条理清晰。在内容介绍过程中，配合讲解的内容穿插相关的问题并要求读者回答，帮助读者自行检查对知识的掌握程度。每章最后都有数目不等的练习及项目，教师可选择使用。本书提供了大量的代码，并对代码进行了详细的介绍，这些代码对学生理解课程内容非常有益。有些未全部实现的类及方法作为课后练习及项目，要求学生完成，书中提供的这些代码成为必要的参考及基础。

在此，非常感谢机械工业出版社华章公司给译者提供的翻译机会。在翻译过程中，译者不仅学习了作者的编写思想，更感受到作者的敬业精神。书中反映出的作者的认真态度，使译者在翻译过程中不敢有丝毫的懈怠。译者还要特别感谢朱勍、张梦玲等编辑。翻译过程中，译者始终得到华章公司温莉芳副总经理、朱勍主任的大力支持和全方位的帮助。责任编辑严格把关，与译者多次探讨重要概念、术语的确切含义及合适的用辞。正是各位编辑的认真负责，才让本书顺利地和读者见面。

本书由辛运伟翻译。翻译过程中，得到了南开大学计算机学院多位老师的 support 和帮助，包括徐敬东教授、刘晓光教授、王刚教授、杨巨峰教授等，由于篇幅所限，恕译者不能一一列出全部人的姓名，在此一并表示衷心的感谢。还要特别感谢国防科技大学计算机学院熊岳山教授、东南大学计算机学院姜浩教授、北京理工大学网络服务中心陈朔鹰主任，正是他们为译者提出的众多非常好的建议，帮助译者顺利完成了本书的翻译。

虽然译者在翻译时非常认真努力，期望以尽量高的水平将本书呈现给读者，但限于译者的水平，很多地方并不能完全体现作者的原意。第 5 版的翻译中也尽量改正了第 4 版翻译中的错误，但书中仍难免有错误之处，敬请广大读者指正。您的任何意见和建议都能帮助进一步完善本书。

感谢尊敬的读者选择了本书。

译者

2019 年 7 月于南开津南园

前　　言

Data Structures and Abstractions with Java, Fifth Edition

欢迎使用第 5 版，本书可作为数据结构课程的教材，例如 CS-2 课程。

作者集 30 余年讲授本科生计算机科学的教学经验，时刻谨记师生的需求来写作本书。作者想让本书适合读者阅读，这样学生学得更容易，老师教得更有效果。为此，本书将资料分为一个个的小部分——我们称之为“段”，这样容易理解且方便学习。模仿现实世界情形的一些例子作为新素材的背景，帮助学生理解抽象概念。使用很多简单的图来解释及阐述复杂的思想。

这次修订重点关注各种数据结构的规范说明和实现的设计决策，并强调安全可靠的程序设计实践。

本书的组织结构

本书本着让教学和学习更容易的宗旨来组织、排列及划分所讨论的主题，主要包括：

- 每次将读者的注意力集中在一个概念上。
- 为读者提供灵活的阅读次序。
- 明确区分抽象数据类型（ADT）的规范说明及其实现。
- 将 Java 的相关内容独立为 Java 插曲，读者可以根据需要选择阅读。

为此，我们将内容分为 30 章，每章由带编号且每次只涉及一个概念的小段组成。每章涉及 ADT 的规范说明及用法或者其实现方式。你可以选择学习一种 ADT 的规范说明，然后研究其实现方式，也可以在考虑实现问题之前学习几种 ADT 的规范说明及用法。本书的组织方式方便你按喜欢的次序选择阅读章节。

我们使用 Java 插曲来介绍与 Java 相关的内容，从而明确地将涉及数据结构的内容与 Java 的具体问题区分开来。这些插曲根据需要穿插在本书的章节之间。不过，我们的关注点是数据结构而不是 Java。从后面的目录中你可以看到这些插曲的标题，以及它们在章节之间的位置。

本版的新内容

第 5 版加强了第 4 版的内容，承袭了原有版本适合入门级学生的教学方法，保留了以前版本中的形式。根据读者的建议及我们自己的愿望，我们对本书做了全面修订，使内容更清晰和准确。本版中的主要修订如下：

- 增加了讨论更多递归内容的一章（第 14 章），介绍语法、语言及回溯。
- 继续介绍安全可靠的编程实践。
- 增加了新的设计决策、注、安全说明及程序设计技巧。
- 在大部分章节中，增加了侧重于游戏、电子商务及财务的新练习和程序设计项目。
- 调整了某些主题的次序。
- 完善了术语、描述和用词，以便于理解。
- 修改了插图，使之更容易阅读和理解。

- 将“自测题”改名为“学习问题”，并将答案移至网站中，鼓励小组一起讨论答案。
- 书中包含了关于 Java 类的附录，而不是将其放在网站中。
- 减少了书中的 Java 代码数量。
- 确保所有的 Java 代码兼容 Java 9。

联系我们

欢迎使用本书的师生联系我们。非常感谢您的意见、建议及校正。我们的联系方式如下：

- 电子邮件：carrano@acm.org 或 timhenry@acm.org
- Twitter：twitter.com/makingCSreal
- 网站：www.makingCSreal.com 和 timothyhenry.net
- Facebook：www.facebook.com/makingCSreal

如何学习本书

本书讨论的内容涉及数据的不同组织方法，以便所给的应用程序能以高效的方式访问并处理数据。这些内容是你未来进一步学习计算机科学知识所不可或缺的，因为它们是创建复杂、可靠的软件所必需的基础知识。不论你是对设计视频游戏感兴趣，还是对设计机器人控制手术的软件感兴趣，学习数据结构都是成功的必经之路。即使你现在不学完本书的全部内容，以后也可能遇到相关话题。我们希望你享受阅读本书的过程，希望本书能成为你未来课程学习的有用参考资料。

读过前言后，你应该读导论，从而快速了解本书要讨论哪些内容，以及开始学习之前你必须了解 Java 的哪些方面。序言中讨论了类的设计及 Java 接口的使用。本书从头至尾使用了接口。附录 A、B 和 C 讨论了 javadoc 注释、Java 类和继承。Java 插曲贯穿于书中，涵盖所需的 Java 高级特性。注意，在封二给出了 Java 保留字及运算符优先级，封三给出了基本数据类型及 Unicode 字符编码。

请一定要浏览前言的其他部分，了解有助于你学习的特点。

提高学习效率的特点

每章的开头是先修章节列表和读完本章要达成的学习目标。贯穿全书的其他教学要素如下：



注 重要的思想用突出显示的段落来表示或概括，意味着要与上下文一起阅读。



安全说明 这个特点介绍并突出显示进行安全可靠的程序设计的方方面面。



问题求解 大的示例以“问题求解”形式给出。提出问题，然后讨论、设计并实现解决方案。



设计决策 为了让读者明了制订一个方案时所做的设计选择，“设计决策”要素一一说明这样的选择，以及具体示例所做选择背后的关系。这些讨论常出现在“问题求解”部分。



示例 说明新概念的众多示例。



程序设计技巧 提出改善或方便程序设计的建议。



学习问题 每章都提出了学习问题，它们与正文混在一起，强化刚介绍的概念。学习问题有助于读者理解本书内容，因为回答它们需要深思。建议在查阅网站中给出的答案前，先和其他人讨论这些问题及其答案。

练习和程序设计项目 求解每章最后的练习和程序设计项目可获得更多的练习机会。很遗憾，我们不能为读者提供这些练习和程序设计项目的答案，只有采用本教材的教师可从出版商那里得到部分答案。要想获得这些练习和项目的帮助，请联系你的老师。

教师及学生资源

从出版商的网站 pearsonhighered.com/carrano 中，可得到下列资料：

- 本书中出现的 Java 代码。
- 本书出版后发现的印刷错误的链接。
- 下面描述的在线内容的链接。

关于配套网站资源，大部分需要访问码，访问码只有原英文版提供，中文版无法使用。

教师资源^①

采用本书的教师可访问 pearsonhighered.com/carrano，登录 Pearson 的教师资源中心（Instructor Resource Center, IRC），得到下列受保护的资料：

- 教师指南。
- PPT 课件，附带用于所有图像的 ADA 兼容的描述文本。
- 教师答案手册。
- 实验手册和解决方案。
- 给教师的 Java 源代码。
- 书中的图。
- 测试用例库。

请联系 Pearson 销售代表获取教师访问码，从 pearsonhighered.com/replocator 可得到联系方式。

学生资源^②

登录出版商的网站 pearsonhighered.com/carrano，可得到下列资料：

- Java 基础（补充材料 1）。
- 文件输入输出（补充材料 2）。
- 词汇表（补充材料 3）。

① 关于本书教辅资源，只有使用本书作为教材的教师才可以申请，需要的教师请联系机械工业出版社华章公司，电话 010-88378991，邮箱 wangguang@hzbook.com。——编辑注
② 中文版的补充材料可以从华章公司网站 (www.hzbook.com) 下载。

- 学习问题答案（补充材料 4）。

注意，从 docs.oracle.com/javase/9/docs/api/ 可得到 Java 类库文档。

内容概要

本书的读者应当已经学完了程序设计课程，学习过 Java 就更好了。附录和在线补充材料涵盖了我们假定读者已经了解的 Java 的基本内容。可以使用这些资料来复习，或者作为从另一种程序设计语言转向 Java 的基础。

- **导论：**首先为将要研究的数据组织做准备，看一些日常的例子。
- **序言：**序言讨论了类的设计。我们讨论的主题包括前置条件、后置条件、断言、接口和统一建模语言（UML）。设计是我们要讲述的一个重要方面。
- **第 1～3 章：**将包作为抽象数据类型（ADT）来介绍。明确地将包的规范说明、使用及实现分为几章来介绍。例如，第 1 章给出了包的规范说明，并提供了几个使用示例。这一章还介绍了 ADT 集合（set）。第 2 章涵盖了使用数组的实现方式，而第 3 章介绍了结点链表，并将其用在包的类定义中。

采用类似的方式，本书中在讨论其他各种 ADT 时也将规范说明与实现分开。可以选择先阅读 ADT 规范说明及用法的章节，之后再阅读实现 ADT 的章节；或者可以按章节出现的次序来阅读，学习了每个 ADT 的规范说明及用法之后马上学习它们的实现。前言后面的先修章节列表能帮助你制订本书的学习计划。

第 2 章不仅仅介绍了 ADT 包的简单实现，还介绍了首先关注核心方法的类的实现过程。当定义类时，先实现并测试这些核心方法，稍后再完成其他方法的定义，这种方式很有效。第 2 章还介绍了安全可靠的程序设计的概念，展示如何在代码中增加这层保护。

- **Java 插曲 1 和 Java 插曲 2：**Java 插曲 1 介绍了泛型，故我们可以将它用在第一个 ADT（即包）中。这个插曲紧接在第 1 章之后。Java 插曲 2 介绍了异常，它紧接在第 2 章之后。使用这些内容来实现 ADT 包。
- **第 4 章：**本章介绍算法的效率和复杂度，这个话题在后面的章节中都有涉及。
- **第 5 和 6 章：**第 5 章讨论栈，给出使用的示例。第 6 章使用数组、结点链表和向量来实现栈。
- **Java 插曲 3：**该插曲展示程序员如何写新的异常类。为此，介绍了如何派生一个已有的异常类。该插曲还介绍了 `finally` 块。
- **第 7 和 8 章：**第 7 章讨论队列、双端队列和优先队列，而第 8 章讨论它们的实现。第 8 章还介绍了循环链表及双向链表，用到了程序员定义的类 `EmptyQueueException`。
- **第 9 章：**接下来，介绍作为问题求解工具的递归，以及它与栈的关系。递归与算法效率也是贯穿全书一再讨论的话题。
- **第 10～12 章：**接下来的三章介绍 ADT 线性表。抽象地讨论集合（collection），然后使用数组和结点链表来实现它。
- **Java 插曲 4 和第 13 章：**Java 迭代器所涉及的内容是标准接口 `Iterator`、`Iterable` 和 `ListIterator`。第 13 章则展示实现这些 ADT 线性表的迭代器方法，讨论并实现了 Java 迭代器接口 `Iterator` 和 `ListIterator`。
- **第 14 章：**这个新章节介绍了更多的递归内容，包括语言、语法和回溯。
- **Java 插曲 5：**该插曲提供了所介绍的排序方法要用到的 Java 概念，介绍标准接口

`Comparable`、泛型方法、限定的类型参数和通配符。

- **第 15 和 16 章：**接下来的两章讨论不同的排序技术及与它们相关的复杂度。考虑这些算法的迭代及递归实现版本。
- **Java 插曲 6：**该插曲讨论可变和不可变对象，这是与前几章关于排序的内容及下一章关于有序表的内容相关的话题。
- **第 17 和 18 章及 Java 插曲 7：**继续线性表的讨论，第 17 章介绍有序表，讨论两种可能的实现方法及它们的效率。第 18 章展示如何使用线性表作为有序表的超类（或父类），并讨论超类的一般设计原则。虽然继承是在附录 C 中概括的，但继承的相关内容（包括保护访问、抽象类及抽象方法）在第 18 章之前的 Java 插曲 7 中介绍。
- **第 19 章：**接下来讨论使用数组或链表保存线性表或有序表的查找策略。这些讨论是后续章节的良好基础。
- **Java 插曲 8：**开始下一章之前，快速学习该插曲中所涉及的多个泛型数据类型的内容是有必要的。
- **第 20 ~ 23 章：**第 20 章讨论 ADT 字典的规范说明及使用。第 21 章介绍使用链表或数组实现字典。第 22 章介绍散列方法，而第 23 章使用散列来实现字典。
- **第 24 和 25 章：**第 24 章讨论树及可能的应用。在树的几个应用示例中介绍二叉查找树及堆。第 25 章介绍二叉树和一般树的实现。
- **Java 插曲 9：**Java 插曲 9 讨论克隆。我们克隆一个数组、一个结点链表及一个二叉结点，还讨论了有序表的克隆。虽然这些内容很重要，但可作为选修内容，因为它们不是后续章节所必需的。
- **第 26 ~ 28 章：**第 26 章重点讨论二叉查找树的实现。第 27 章展示如何使用数组来实现堆。第 28 章介绍平衡查找树，还介绍 AVL 树、2-3 树、2-4 树、红黑树及 B 树。
- **第 29 和 30 章：**最后，我们讨论图，学习几个应用及两种实现方式。
- **附录 A ~ 附录 C：**附录中是 Java 内容的补充。如前所述，附录 A 涉及程序设计风格和注释，介绍了 `javadoc` 注释，定义了本书中使用的标签。附录 B 讨论了 Java 类，而附录 C 扩展了这个话题，讨论组成与继承。

先修章节

每一章和附录都假定读者肯定已经学习了之前的内容。下表列出这些先修章节，数字表示章号，字母表示附录，每个 Java 插曲编号前面冠以“JI”。读者可以用这些信息制订本书的学习计划。

| 章号 | 章名 | 先修章节 |
|-----------|-----------|-----------------|
| 序言 | 设计类 | 补充材料 1, A, B, C |
| 第 1 章 | 包 | 序言, C |
| Java 插曲 1 | 泛型 | 序言 |
| 第 2 章 | 使用数组实现包 | 序言, 1 |
| Java 插曲 2 | 异常 | 补充材料 1, B, C |
| 第 3 章 | 使用链式数据实现包 | 1, 2, JI2 |
| 第 4 章 | 算法的效率 | B, 2, 3 |
| 第 5 章 | 栈 | 序言, 1, JI2 |

(续)

| 章号 | 章名 | 先修章节 |
|-----------|-----------------|-------------------------------|
| 第 6 章 | 栈的实现 | 2, 3, 4, 5 |
| Java 插曲 3 | 再论异常 | C, JI2 |
| 第 7 章 | 队列、双端队列和优先队列 | 序言, 5 |
| 第 8 章 | 队列、双端队列和优先队列的实现 | 2, 3, 6, 7 |
| 第 9 章 | 递归 | B, 2, 3, 4, 5 |
| 第 10 章 | 线性表 | 导论, B, 序言, JI2, 1 |
| 第 11 章 | 使用数组实现线性表 | 序言, 2, 4, 10 |
| 第 12 章 | 使用链式数据实现线性表 | 3, 8, 10, 11 |
| Java 插曲 4 | 迭代器 | JI2, 10 |
| 第 13 章 | ADT 线性表的迭代器 | 11, 12, JI4 |
| 第 14 章 | 使用递归求解问题 | 5, 9 |
| Java 插曲 5 | 再论泛型 | JI1 |
| 第 15 章 | 排序简介 | 3, 4, 9, JI5 |
| 第 16 章 | 更快的排序方法 | 4, 9, JI5, 15 |
| Java 插曲 6 | 可变及不可变对象 | C, 10 |
| 第 17 章 | 有序表 | 4, 9, 10, 12, JI6 |
| Java 插曲 7 | 继承和多态 | 序言, C, 6 |
| 第 18 章 | 继承和线性表 | C, 10, 11, 12, 17, JI7 |
| 第 19 章 | 查找 | 4, 9, 10, 11, 12, 17 |
| Java 插曲 8 | 三论泛型 | B, JI5 |
| 第 20 章 | 字典 | 10, JI4, 13, 19, JI8 |
| 第 21 章 | 字典的实现 | 3, 4, 10, 11, 12, JI4, 19, 20 |
| 第 22 章 | 散列简介 | 20, 21 |
| 第 23 章 | 使用散列实现字典 | 4, 11, 12, JI4, 20, 21, 22 |
| 第 24 章 | 树 | 5, 9, 12, JI4, 19 |
| 第 25 章 | 树的实现 | C, JI2, 5, 7, 12, 24 |
| Java 插曲 9 | 克隆 | JI5, JI6, 17, 25, B, C |
| 第 26 章 | 二叉查找树的实现 | C, 9, 20, 24, 25 |
| 第 27 章 | 堆的实现 | 2, 11, 24 |
| 第 28 章 | 平衡查找树 | 24, 25, 26 |
| 第 29 章 | 图 | 5, 7, 24 |
| 第 30 章 | 图的实现 | 5, 7, 10, JI4, 13, 20, 24, 29 |
| 附录 A | 文档和程序设计风格 | Java 的一些知识 |
| 附录 B | Java 类 | 补充材料 1 |
| 附录 C | 从其他类创建类 | B |
| 补充材料 1 | Java 基础 | 程序设计语言知识 |
| 补充材料 2 | 文件输入输出 | 序言, JI2, 补充材料 1 |

致谢

衷心感谢以下审稿人认真阅读第 4 版，直言不讳地提出意见和建议，从而大大改进了我们的工作：

Prakash Duraisamy——迈阿密大学

David Gries——康奈尔大学

Hakam Alomari——迈阿密大学
Jack Pope——北京内平社区学院
Anna Rafferty——卡尔顿大学

特别感谢 Pearson 教育集团在本书漫长的修订期间对我们的支持和帮助：执行项目经理 Tracy Johnson、项目管理助理 Meghan Jacoby、管理内容制作人 Scott Disanno 和内容制作人 Lora Friedenthal。长期合作的文字编辑 Rebecca Pepper 确保本书清楚、正确，且语言无误。非常感谢大家！

除了前面提到的人之外，还要感谢其他许多人提供的帮助。Tim Henry 为本版制作了 PPT 课件。俄克拉荷马城市大学的 Charles Hoot 教授编写了实验手册，阿克伦大学的 Kathy Liszka 教授编写了测试问题集，Joe Erickson 和 Jesse Grabowski 提供了很多程序设计项目的答案。再次感谢本书之前版本的审稿人。

第 4 版审稿人：

Tony Allevato——弗吉尼亚理工学院暨州立大学
Mary Boelk——马凯特大学
Suzanne Buchele——西南大学
Kevin Buffardi——弗吉尼亚理工学院暨州立大学
Jose Cordova——路易斯安那门罗大学
Greg Gagne——威斯敏斯特学院
Victoria Hilford——休斯敦大学
Jim Huggins——凯特林大学
Shamim Kahn——哥伦布州立大学
Kathy Liszka——阿克伦大学
Eli Tilevich——弗吉尼亚理工学院暨州立大学
Jianhua Yang——哥伦布州立大学
Michelle Zhu——南伊利诺伊大学

第 3 版审稿人：

Steven Andrianoff——圣文德大学
Brent Baas——拉托诺大学
Timothy Henry——新英格兰理工学院
Ken Martin——北佛罗里达大学
Bill Siever——西北密苏里州立大学
Lydia Sinapova——辛普森学院
Lubomir Stanchev——印第安纳大学
Judy Walters——中北学院
Xiaohui Yuan——北得克萨斯大学

第 2 版审稿人：

Harold Anderson——曼利斯特学院
Razvan Andonie——华盛顿中心大学
Tom Blough——伦斯勒理工学院

Chris Brooks——旧金山大学

Adrienne Decker——纽约州立大学布法罗分校

Henry Etlinger——罗切斯特理工学院

Derek Harter——得克萨斯 A&M 大学

Timothy Henry——新英格兰理工学院

Robert Holloway——威斯康星大学麦迪逊分校

Charles Hoot——俄克拉荷马城市大学

Teresa Leyk——得克萨斯 A&M 大学

Robert McGinn——南伊利诺伊大学卡本代尔校区

Edward Medvid——玛丽蒙特大学

Charles Metzler——旧金山城市学院

Daniel Zeng——亚利桑那大学

第 1 版审稿人：

David Boyd——瓦尔多斯塔州立大学

Dennis Brylow——普渡大学

Michael Croswell——产业培训师 / 顾问

Matthew Dickerson——明德学院

Robert Holloway——威斯康星大学麦迪逊分校

John Motil——加州州立大学北岭分校

Bina Ramamurthy——纽约州立大学布法罗分校

David Surma——瓦尔帕莱索大学

还要感谢在前几版写作过程中帮助过我们的很多人。他们是：Alan Apt, Steve Armstrong, James Blanding, Lianne Dunn, Bob Englehardt, Mike Giacobbe, Toni Holm, Charles Hoot, Brian Jepson, Rose Kernan, Christianna Lee, Patrick Lindner, John Lovell, Vince O'Brien, Patty Roy, Walt Savitch, Ben Schomp, Heather Scott, Carole Snyder, Chirag Thakkar, Camille Trentacoste, Nate Walker, Xiaohong Zhu。

最后，感谢我们的家人和朋友 Doug、Joanne、Tita、Bobby、Ted、Nancy、Sue、Tom、Bob、Maybeth、Marge 和 Lorraine 给了我们远离计算机时的生活。

谢谢你们每一个人，谢谢你们的专业和鼓励。

Frank M. Carrano

Timothy M. Henry

目 录

Data Structures and Abstractions with Java, Fifth Edition

| | |
|----------------------|----------|
| 出版者的话 | |
| 译者序 | |
| 前言 | |
| 导论 组织数据 | <i>I</i> |
| 序言 设计类 | 3 |
| 封装 | 3 |
| 规范说明方法 | 5 |
| 注释 | 5 |
| 前置条件和后置条件 | 5 |
| 断言 | 6 |
| Java 接口 | 7 |
| 写一个接口 | 8 |
| 实现一个接口 | 9 |
| 接口作为数据类型 | 11 |
| 派生一个接口 | 12 |
| 接口内的命名常量 | 13 |
| 选择类 | 14 |
| 标识类 | 15 |
| CRC 卡 | 15 |
| 统一建模语言 | 16 |
| 重用类 | 18 |
| 练习 | 19 |
| 项目 | 19 |
| 第 1 章 包 | 22 |
| 什么是包 | 22 |
| 包的行为 | 23 |
| 规范说明一个包 | 23 |
| 一个接口 | 28 |
| 使用 ADT 包 | 30 |
| 像使用自动贩卖机一样使用 ADT | 33 |
| ADT 集合 | 34 |
| Java 类库：接口 Set | 35 |
| 本章小结 | 35 |
| 练习 | 36 |
| 项目 | 37 |
| Java 插曲 1 泛型 | 40 |
| 泛型数据类型 | 40 |
| 接口中的泛型 | 40 |
| 泛型类 | 41 |
| 第 2 章 使用数组实现包 | 44 |
| 使用定长数组实现 ADT 包 | 44 |
| 模拟 | 44 |
| 一组核心方法 | 45 |
| 实现核心方法 | 46 |
| 让实现安全 | 52 |
| 测试核心方法 | 54 |
| 实现更多的方法 | 57 |
| 删除项的方法 | 59 |
| 使用变长数组实现 ADT 包 | 67 |
| 变长数组 | 67 |
| 包的新实现 | 69 |
| 使用数组实现 ADT 包的优缺点 | 71 |
| 本章小结 | 72 |
| 程序设计技巧 | 72 |
| 练习 | 73 |
| 项目 | 74 |
| Java 插曲 2 异常 | 75 |
| 基础 | 75 |
| 处理异常 | 77 |
| 延缓处理：throws 子句 | 77 |
| 现在处理：try-catch 块 | 78 |
| 多个 catch 块 | 79 |
| 抛出异常 | 80 |

| | |
|--------------------------------|-----|
| 第 3 章 使用链式数据实现包 | 83 |
| 链式数据 | 83 |
| 添加到开头形成一个链表 | 84 |
| ADT 包的链式实现 | 85 |
| 私有类 Node | 85 |
| 类 LinkedBag 的框架 | 87 |
| 定义一些核心方法 | 88 |
| 测试核心方法 | 91 |
| 方法 getFrequencyOf | 92 |
| 方法 contains | 93 |
| 从链表中删除一项 | 94 |
| 方法 remove 和 clear | 95 |
| 有设置和获取方法的类 Node | 98 |
| 使用链表实现 ADT 包的优缺点 | 101 |
| 本章小结 | 101 |
| 程序设计技巧 | 101 |
| 练习 | 102 |
| 项目 | 102 |
| 第 4 章 算法的效率 | 104 |
| 动机 | 104 |
| 算法效率的衡量 | 105 |
| 统计基本操作 | 107 |
| 最优、最差和平均情况 | 109 |
| 大 O 表示 | 109 |
| 程序结构的复杂度 | 112 |
| 图示化效率 | 113 |
| 实现 ADT 包的效率 | 115 |
| 基于数组的实现 | 115 |
| 链式实现 | 117 |
| 两种实现的比较 | 118 |
| 本章小结 | 118 |
| 练习 | 118 |
| 项目 | 121 |
| 第 5 章 栈 | 123 |
| ADT 栈的规范说明 | 123 |
| 使用栈来处理代数表达式 | 127 |
| 问题求解：检查中缀代数表达式中 平衡的分隔符 | 128 |
| 问题求解：将中缀表达式转换为 后缀表达式 | 132 |
| 问题求解：计算后缀表达式的值 | 136 |
| 问题求解：计算中缀表达式的值 | 137 |
| 程序栈 | 139 |
| Java 类库：类 Stack | 140 |
| 本章小结 | 141 |
| 程序设计技巧 | 141 |
| 练习 | 141 |
| 项目 | 143 |
| 第 6 章 栈的实现 | 146 |
| 链式实现 | 146 |
| 基于数组的实现 | 149 |
| 基于向量的实现 | 152 |
| Java 类库：类 Vector | 153 |
| 使用 Vector 实现 ADT 栈 | 154 |
| 本章小结 | 155 |
| 练习 | 156 |
| 项目 | 156 |
| Java 插曲 3 再论异常 | 158 |
| 程序员定义的异常类 | 158 |
| 继承和异常 | 162 |
| finally 块 | 163 |
| 第 7 章 队列、双端队列和优先 队列 | 166 |
| ADT 队列 | 166 |
| 问题求解：模拟排队 | 169 |
| 问题求解：计算股票售出的资本 收益 | 174 |
| Java 类库：接口 Queue | 177 |
| ADT 双端队列 | 177 |
| 问题求解：计算股票售出的资本 收益 | 180 |
| Java 类库：接口 Deque | 181 |
| Java 类库：类 ArrayDeque | 182 |
| ADT 优先队列 | 183 |
| 问题求解：跟踪指派 | 184 |

| | | | |
|------------------------------------|------------|---------------------------------|------------|
| Java 类库：类 PriorityQueue | 185 | 使用 ADT 线性表 | 246 |
| 本章小结 | 186 | 问题求解：使用大整数 | 250 |
| 程序设计技巧 | 187 | Java 类库：接口 List | 252 |
| 练习 | 187 | Java 类库：类 ArrayList | 252 |
| 项目 | 188 | 本章小结 | 253 |
| 第 8 章 队列、双端队列和优先队列的实现 | 192 | 练习 | 253 |
| 队列的链式实现 | 192 | 项目 | 254 |
| 基于数组实现队列 | 196 | 第 11 章 使用数组实现线性表 | 257 |
| 循环数组 | 196 | 使用数组实现 ADT 线性表 | 257 |
| 带一个未用元素的循环数组 | 198 | 模拟 | 257 |
| 队列的循环链式实现 | 203 | Java 实现 | 259 |
| 两部分组成的循环链表 | 204 | 使用数组实现 ADT 线性表的效率 | 266 |
| Java 类库：类 AbstractQueue | 209 | 本章小结 | 268 |
| 队列的双向链式实现 | 209 | 练习 | 268 |
| 优先队列的可能实现方案 | 213 | 项目 | 269 |
| 本章小结 | 213 | 第 12 章 使用链式数据实现线性表 | 271 |
| 程序设计技巧 | 214 | 结点链表上的操作 | 271 |
| 练习 | 214 | 在不同的位置添加结点 | 271 |
| 项目 | 215 | 从不同的位置删除结点 | 275 |
| 第 9 章 递归 | 217 | 私有方法 getNodeAt | 276 |
| 什么是递归 | 217 | 实现之初 | 277 |
| 跟踪递归方法 | 221 | 数据域和构造方法 | 278 |
| 返回一个值的递归方法 | 224 | 添加到线性表表尾 | 279 |
| 递归处理数组 | 226 | 在线性表的给定位置添加 | 280 |
| 递归处理链表 | 228 | 方法 isEmpty 和 toArray | 281 |
| 递归方法的时间效率 | 230 | 测试核心方法 | 283 |
| countDown 的时间效率 | 230 | 继续实现 | 284 |
| 计算 x^n 的时间效率 | 231 | 完善实现 | 286 |
| 尾递归 | 232 | 尾引用 | 287 |
| 使用栈来替代递归 | 233 | 使用链表实现 ADT 线性表的效率 | 290 |
| 本章小结 | 234 | Java 类库：类 LinkedList | 292 |
| 程序设计技巧 | 234 | 本章小结 | 292 |
| 练习 | 235 | 练习 | 293 |
| 项目 | 237 | 项目 | 294 |
| 第 10 章 线性表 | 240 | Java 插曲 4 迭代器 | 296 |
| ADT 线性表的规范说明 | 240 | 什么是迭代器 | 296 |
| | | 接口 Iterator | 297 |

| | | | |
|---------------------------------|-----|-----------------------------|-----|
| 接口 Iterable | 298 | 通配符 | 357 |
| 使用接口 Iterator | 299 | 限定的通配符 | 357 |
| Iterable 和 for-each 循环 | 302 | | |
| 接口 ListIterator | 303 | 第 15 章 排序简介 | 360 |
| 再论接口 List | 306 | 对数组进行排序的 Java 方法的组织 | 360 |
| 使用接口 ListIterator | 306 | 选择排序 | 361 |
| 第 13 章 ADT 线性表的迭代器 | 309 | 迭代的选择排序 | 362 |
| 实现迭代器的方法 | 309 | 递归的选择排序 | 364 |
| 独立类迭代器 | 309 | 选择排序的效率 | 364 |
| 内部类迭代器 | 312 | 插入排序 | 365 |
| 链式实现 | 313 | 迭代的插入排序 | 366 |
| 基于数组的实现 | 316 | 递归的插入排序 | 367 |
| 为什么迭代器方法在它自己的类中 | 319 | 插入排序的效率 | 369 |
| 基于数组实现接口 ListIterator | 320 | 结点链表上的插入排序 | 369 |
| 内部类 | 322 | 希尔排序 | 372 |
| 本章小结 | 327 | 算法 | 373 |
| 程序设计技巧 | 327 | 希尔排序的效率 | 374 |
| 练习 | 327 | 算法比较 | 374 |
| 项目 | 329 | 本章小结 | 375 |
| 第 14 章 使用递归求解问题 | 331 | 程序设计技巧 | 375 |
| 困难问题的简单求解方案 | 331 | 练习 | 375 |
| 简单问题的低劣求解方案 | 336 | 项目 | 377 |
| 语言和语法 | 338 | | |
| Java 标识符语言 | 338 | 第 16 章 更快的排序方法 | 379 |
| 前缀表达式语言 | 339 | 归并排序 | 379 |
| 前缀表达式的计算 | 342 | 归并数组 | 379 |
| 间接递归 | 343 | 递归的归并排序 | 380 |
| 回溯 | 343 | 归并排序的效率 | 382 |
| 穿越迷宫 | 344 | 迭代的归并排序 | 384 |
| n 皇后问题 | 345 | Java 类库中的归并排序 | 384 |
| 本章小结 | 347 | 快速排序 | 385 |
| 练习 | 348 | 快速排序的效率 | 385 |
| 项目 | 349 | 创建划分 | 386 |
| Java 插曲 5 再论泛型 | 352 | 实现快速排序 | 388 |
| 接口 Comparable | 352 | Java 类库中的快速排序 | 390 |
| 泛型方法 | 354 | 基数排序 | 390 |
| 限定的类型参数 | 354 | 基数排序的伪代码 | 392 |
| | | 基数排序的效率 | 392 |
| | | 算法比较 | 393 |
| | | 本章小结 | 393 |

| | | | |
|---------------------------|------------|---------------------------------------|-----|
| 练习 | 394 | 第 19 章 查找 | 443 |
| 项目 | 395 | 问题 | 443 |
| Java 插曲 6 可变及不可变对象 | 397 | 在无序数组中查找 | 443 |
| 可变对象 | 397 | 无序数组中的迭代顺序查找 | 444 |
| 不可变对象 | 398 | 无序数组中的递归顺序查找 | 445 |
| 创建只读类 | 399 | 顺序查找数组的效率 | 446 |
| 伴生类 | 400 | 在有序数组中查找 | 446 |
| 第 17 章 有序表 | 403 | 有序数组中的顺序查找 | 447 |
| ADT 有序表的规范说明 | 403 | 有序数组中的二分查找 | 447 |
| 使用 ADT 有序表 | 406 | Java 类库: <code>binarySearch</code> 方法 | 451 |
| 链式实现 | 407 | 数组中二分查找的效率 | 451 |
| 方法 <code>add</code> | 408 | 在无序链表中查找 | 453 |
| 链式实现的效率 | 414 | 无序链表中的迭代顺序查找 | 453 |
| 使用 ADT 线性表的实现 | 414 | 无序链表中的递归顺序查找 | 453 |
| 效率问题 | 417 | 链表中顺序查找的效率 | 454 |
| 本章小结 | 418 | 在有序链表中查找 | 454 |
| 练习 | 419 | 有序链表中的顺序查找 | 454 |
| 项目 | 419 | 有序链表中的二分查找 | 455 |
| Java 插曲 7 继承和多态 | 421 | 查找方法的选择 | 455 |
| 继承的其他方面 | 421 | 本章小结 | 456 |
| 何时使用继承 | 421 | 程序设计技巧 | 456 |
| 保护访问 | 422 | 练习 | 456 |
| 抽象类和方法 | 422 | 项目 | 458 |
| 接口与抽象类 | 424 | Java 插曲 8 三论泛型 | 460 |
| 多态 | 425 | 多个泛型 | 460 |
| 第 18 章 继承和线性表 | 430 | 第 20 章 字典 | 462 |
| 使用继承实现有序表 | 430 | ADT 字典的规范说明 | 462 |
| 设计一个基类 | 432 | Java 接口 | 465 |
| 创建抽象基类 | 436 | 迭代器 | 466 |
| 有序表的高效实现 | 439 | 使用 ADT 字典 | 468 |
| 方法 <code>add</code> | 439 | 问题求解: 电话号码簿 | 468 |
| 本章小结 | 440 | 问题求解: 单词的频率 | 472 |
| 程序设计技巧 | 440 | 问题求解: 单词的索引 | 475 |
| 练习 | 440 | Java 类库: 接口 <code>Map</code> | 478 |
| 项目 | 441 | 本章小结 | 478 |

| | | | |
|--------------------------|-----|--------------------|-----|
| 项目 | 479 | 迭代器 | 525 |
| 第 21 章 字典的实现 | 482 | Java 类库：类 HashMap | 526 |
| 基于数组的实现 | 482 | Java 类库：类 HashSet | 527 |
| 基于无序数组的字典 | 482 | 本章小结 | 527 |
| 基于有序数组的字典 | 487 | 练习 | 528 |
| 链式实现 | 491 | 项目 | 528 |
| 无序链式字典 | 491 | 第 24 章 树 | 531 |
| 有序链式字典 | 492 | 树的概念 | 531 |
| 本章小结 | 494 | 层次结构 | 531 |
| 程序设计技巧 | 494 | 树的术语 | 532 |
| 练习 | 494 | 树的遍历 | 537 |
| 项目 | 495 | 二叉树的遍历 | 537 |
| 第 22 章 散列简介 | 496 | 一般树的遍历 | 538 |
| 什么是散列 | 496 | 用于树的 Java 接口 | 539 |
| 散列函数 | 498 | 用于所有树的接口 | 539 |
| 计算散列码 | 498 | 用于二叉树的接口 | 540 |
| 将散列码压缩为散列表的下标 | 501 | 二叉树示例 | 541 |
| 解决冲突 | 502 | 表达式树 | 541 |
| 开放地址的线性探查 | 502 | 决策树 | 543 |
| 开放地址的二次探查 | 507 | 二叉查找树 | 545 |
| 开放地址的双散列 | 508 | 堆 | 547 |
| 开放地址的潜在问题 | 510 | 一般树示例 | 548 |
| 拉链法 | 510 | 解析树 | 549 |
| 本章小结 | 513 | 游戏树 | 550 |
| 练习 | 513 | 本章小结 | 550 |
| 项目 | 514 | 练习 | 551 |
| 第 23 章 使用散列实现字典 | 516 | 项目 | 553 |
| 散列的效率 | 516 | 第 25 章 树的实现 | 555 |
| 装填因子 | 516 | 二叉树中的结点 | 555 |
| 开放地址法的代价 | 517 | 二叉结点类 | 556 |
| 拉链法的代价 | 518 | ADT 二叉树的实现 | 557 |
| 再散列 | 519 | 创建基本二叉树 | 557 |
| 冲突解决机制的比较 | 520 | 方法 initializeTree | 559 |
| 使用散列实现字典 | 521 | 访问方法和赋值方法 | 561 |
| 散列表中的项 | 521 | 计算高度和结点个数 | 562 |
| 数据域和构造方法 | 522 | 遍历 | 563 |
| 方法 getValue、remove 和 add | 523 | 表达式树的实现 | 567 |

| | | | |
|------------------------------|------------|-----------------------------|------------|
| 用于一般树的结点 | 568 | 第 27 章 堆的实现 | 620 |
| 使用二叉树表示一般树 | 569 | 再论：ADT 堆 | 620 |
| 本章小结 | 570 | 使用数组表示堆 | 620 |
| 程序设计技巧 | 570 | 添加项 | 623 |
| 练习 | 570 | 删除根 | 626 |
| 项目 | 572 | 创建堆 | 628 |
| Java 插曲 9 克隆 | 574 | 堆排序 | 630 |
| 可克隆的对象 | 574 | 本章小结 | 633 |
| 克隆一个数组 | 579 | 练习 | 633 |
| 克隆一个链表 | 581 | 项目 | 634 |
| 有序表的克隆 | 585 | | |
| 克隆一个二叉结点 | 587 | | |
| 第 26 章 二叉查找树的实现 | 589 | 第 28 章 平衡查找树 | 636 |
| 入门知识 | 589 | AVL 树 | 636 |
| 用于二叉查找树的接口 | 590 | 单旋转 | 636 |
| 重复项 | 591 | 双旋转 | 639 |
| 开始定义类 | 592 | 实现细节 | 642 |
| 查找和获取 | 593 | 2-3 树 | 646 |
| 遍历 | 595 | 在 2-3 树中进行查找 | 646 |
| 添加一项 | 595 | 向 2-3 树中添加项 | 647 |
| 递归实现 | 596 | 添加过程中结点的分裂 | 649 |
| 迭代实现 | 598 | 2-4 树 | 650 |
| 删除一项 | 600 | 向 2-4 树中添加项 | 650 |
| 删除叶结点中的项 | 600 | AVL 树、2-3 树和 2-4 树的比较 | 652 |
| 删除仅有一个孩子的结点中的项 | 600 | 红黑树 | 653 |
| 删除有两个孩子的结点中的项 | 601 | 红黑树的特性 | 654 |
| 删除根中的项 | 604 | 向红黑树中添加项 | 654 |
| 递归实现 | 604 | Java 类库：类 TreeMap | 659 |
| 迭代实现 | 607 | B 树 | 659 |
| 操作效率 | 610 | 本章小结 | 660 |
| 平衡的重要性 | 611 | 练习 | 660 |
| 结点按什么次序添加 | 611 | 项目 | 661 |
| ADT 字典的实现 | 612 | | |
| 本章小结 | 614 | | |
| 练习 | 615 | | |
| 项目 | 617 | | |
| 第 29 章 图 | 663 | | |
| 一些示例及术语 | 663 | | |
| 公路地图 | 663 | | |
| 航空公司的航线 | 665 | | |
| 迷宫 | 666 | | |
| 先修课程 | 666 | | |
| 树 | 667 | | |

| | | | |
|--------------------------|------------|----------------------------|-----|
| 遍历 | 667 | 类 Vertex 的实现 | 691 |
| 广度优先遍历 | 668 | ADT 图的实现 | 694 |
| 深度优先遍历 | 669 | 基本操作 | 694 |
| 拓扑序 | 670 | 图算法 | 697 |
| 路径 | 672 | 本章小结 | 699 |
| 寻找路径 | 673 | 练习 | 699 |
| 无权图中的最短路径 | 673 | 项目 | 701 |
| 带权图中的最短路径 | 675 | | |
| 用于 ADT 图的 Java 接口 | 678 | 附录 A 文档和程序设计风格 | 702 |
| 本章小结 | 681 | 附录 B Java 类 | 706 |
| 练习 | 682 | 附录 C 从其他类创建类 | 727 |
| 项目 | 684 | | |
| 第 30 章 图的实现 | 686 | | |
| 两个实现概述 | 686 | 补充材料 1 Java 基础 (在线) | |
| 邻接矩阵 | 686 | 补充材料 2 文件输入输出 (在线) | |
| 邻接表 | 687 | 补充材料 3 词汇表 (在线) | |
| 顶点和边 | 688 | 补充材料 4 学习问题答案 (在线) | |
| 规范说明类 Vertex | 688 | | |
| 内部类 Edge | 690 | | |

组织数据

环顾四周，就会发现人们安排事情的方式。早晨去商店时，你会站到线后等待付账。这条线会让人们按先来后到的次序排队。线后的第一个人是最先得到服务的人，也是最先离开队列的人。最终，你到达线前，结账后拿着你购买的一包物品离开商店。包里面的东西没有特别的次序，且有些还是一样的。

你看到桌子上的一摞书或一叠纸了吗？很容易看到或拿走一摞东西最上面的一件，或者在一摞东西上面放一件新的东西。一摞东西也是按时间顺序组织的，最后放的在最上面，最先放的在最下面。

在桌子上，可以看到做事清单。清单中的每一项对你或重要或不重要。写这些项时，可能是按照它们的重要性排列的，也可能是按照字母序排列的。这个次序是你定的，清单只是写这些项的地方。

字典是单词及其定义的字母序列表。你在里面查找一个单词从而得到它的定义。如果你的字典是纸制印刷的，则字母序的组织方式能帮助你快速找到这个单词。如果你的字典是电子版的，则字母序的组织方式是隐含的，但它仍然能加快查找过程。

再来看看你的计算机，你的文件是放到文件夹或目录中的。每个文件夹又包含若干其他的文件夹或文件。这种组织类型是层次的。如果将它画成图，会得到一个类似家族树或公司内部部门图的东西。数据的这些组织方式是类似的，称为树。

最后来看看道路地图，你正用它来规划周末游。道路和城市的地图向你展示如何从一个地方到达另一个地方。通常会有几条不同的路。一条路可能更短些，另一条路可能更快些。地图的组织方式称为图。



计算机程序也需要组织它们的数据。其组织方式类似于我们刚刚引用的例子。也就是，程序可以使用栈、线性表、字典等。这些数据组织方式表示为抽象数据类型。抽象数据类型 (Abstract Data Type, ADT) 是描述数据集 (set) 及数据上操作的规范说明。每个 ADT 说明存储什么数据及对数据进行什么操作。因为 ADT 不明示如何保存数据，也不说明如何实现

操作，所以我们可以脱离程序设计语言来谈论 ADT。相比之下，数据结构（data structure）是某种程序设计语言中 ADT 的一种实现。

集合（collection）是个一般术语，是指含有一组对象的 ADT。有些集合允许有重复项，而另一些则不允许有。有些集合按特定的次序排列内容，另一些则没有次序。

我们可以创建一个 ADT 包（bag），它由一个无序集合构成，其中允许有重复值。它像是一个杂货店的袋子、一个午餐袋，或一个薯片袋。假定从薯片袋中拿出一片。你不知道薯片何时放到袋中。你不知道袋子中是否有另外一片，它的形状与刚拿走的那片一模一样。但你真的不在意这个。如果在意，就不会将薯片放到袋子中！

包中的内容没有次序，但有时你想让它们有次序。ADT 可以以多种方式安排项的次序。例如，ADT 线性表（list）对项进行编号。这样，线性表有第一项、第二项，等等。虽然可以将项添加到线性表尾，但也可以将项添加到线性表头，或者在两个项的中间。这样操作后新加项后面的项要重新编号。另外，可以删除线性表指定位置的项。所以线性表中项的位置并不能表明这个项是何时添加进来的。注意，线性表不决定项的放置位置，这件事由你来决定。

相反，ADT 栈（stack）和队列（queue）按时间确定项的次序。当从栈中删除项时，删除的是最后添加的项。当从队列中删除项时，删除的是最早添加的项。所以，栈像是一摞书。你可以拿走最上面的书，或者将另一本书放在这摞书的上面。队列像是一队人。人从队列前头离开，站队时站到最后。

如果项可以进行比较，则有些 ADT 按排序的次序管理项。比方说，字符串可以按字母序组织。例如，当在 ADT 有序线性表（sorted list）中添加项时，由 ADT 来确定这个项在线性表中的位置。你不用指明这个项的位置，但在 ADT 线性表中需要指明。

ADT 字典（dictionary）含有项对，很像是字典中含有的一个单词及其定义。在这个例子中，单词充当关键字（key），用它来查找项。有些字典对项进行排序，有些字典没有排序。

ADT 树（tree）根据层次组织项。例如，在家族树中，人与其孩子和父母相关联。ADT 二叉查找树（binary search tree）结合了层次和排序的方式来组织项，这使得项的查找更容易。

ADT 图（graph）是 ADT 树的推广，它按照项之间的关系而不是层次来组织。例如，道路图是一个图，展示的是城镇之间已有的道路和距离。

本书介绍如何使用并实现这些数据组织方式。本书中假定你已经了解了 Java。不过，全书中称为 Java 插曲的一些特殊段落，集中介绍 Java 的相关方面，这些内容对读者来说可能是全新的，包括如何处理异常。

如果需要复习一下，附录及在线补充材料对你很有用。附录 A 概述了写适用于 javadoc 注释的方法。附录 B 讨论了类和方法的基础结构，而附录 C 介绍了组成和继承的要点。4 个在线补充材料在华章公司网站中提供（www.hzbook.com）。补充材料 1 复习了 Java 中的基本语句，补充材料 2 介绍如何读入及写到外部文件，补充材料 3 是词汇表，补充材料 4 是学习问题答案。可以下载这些补充材料，需要时作为参考。

紧接着在后面的“序言”讨论了如何设计类、规范说明方法及写 Java 接口。使用接口及写注释来说明方法，都是介绍 ADT 不可缺少的部分。

设计类

先修章节：补充材料 1、附录 A、附录 B、附录 C

面向对象程序设计体现了三个设计概念：封装、继承和多态。如果不熟悉继承和多态，请参看附录 A、B 和 C。这里我们讨论封装，这是设计类的过程中隐藏实现细节的一种方式。我们强调，在方法实现之前规范说明它应该有什么行为是重要的，在程序中将规范说明作为注释也是重要的。

我们介绍 Java 接口，这是将类行为的声明与其实现分开的一种方式。最后，介绍用于标识特定解决方案所需的类的一些初级技术。

封装

如果你想学习驾驶，那么对汽车的哪些描述对你最有用？显然不是描述它的发动机的工作过程。当你想学习驾驶时，这样的细节不是必要的。事实上，可以以你的方式获知这些细节。如果你想学习驾驶汽车，最有用的汽车描述是下面这样的特点：

- 如果你将脚踩在油门踏板上，汽车将开得更快。
- 如果你将脚踩在制动踏板上，汽车将慢下来并最终停止。
- 如果你将方向盘向右转，汽车将右转。
- 如果你将方向盘向左转，汽车将左转。

就像你不需要告诉想开车的人发动机是如何工作的一样，你也不需要告诉使用一款软件的人 Java 实现的全部细节。同样，假定你为另一位程序员写了一个用在程序中的软件组件，你应该告诉其他的程序员如何使用它，而不是与程序员分享如何写软件的细节。

封装（encapsulation）是面向对象程序设计的设计原则之一。“封装”这个词听上去好像是把东西放进胶囊，这个想象确实是正确的。封装隐藏了“胶囊”里的细节。由于这个原因，封装常常被称为信息隐藏（information hiding）。但不是所有的事情都应该隐藏。在汽车里，有些东西是可见的——像是踏板和方向盘——而其他的则藏在引擎盖下面。换句话说，汽车是封装的，这样，隐藏了细节，只有驾车所需的控制是可见的，如图 P-1 所示。类似地，你应该封装 Java 代码，让细节隐藏，而只有必需的控制是可见的。

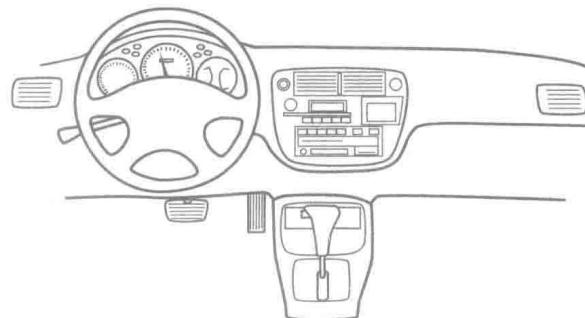


图 P-1 汽车的控制装置对司机是可见的，但它的内部工作机理是隐藏的

封装将数据和方法放到一个类中，而隐藏了使用类时不必要的实现细节。如果类的设计良好，使用它就不需要去理解它的实现。程序员可以在不知道代码细节的情况下使用类的方法。程序员只需要知道如何为方法提供相应的参数，让方法执行正确的动作。简单地说，程序员不必担心类定义的内部细节。使用封装软件来写更多软件的程序员，他的任务更简单。结果，软件生产得更快，错误也更少。

P.2 **注：**封装是面向对象程序设计的设计原则之一，它将数据和方法放到一个类中，故而隐藏了类实现的细节。程序员仅需要知道使用这个类的信息就足够了。设计良好的类，即使每个方法都隐藏了，也能使用。

P.3 **抽象** (abstraction) 是一个要求你关注什么而不是如何的过程。当设计类时，执行的是 **数据抽象** (data abstraction)。你关注你想做的，或关注数据，而不担心如何完成这些任务，及如何表示数据。抽象要求你将注意力集中于哪些数据和操作是重要的。当抽象某件事时，你要确定中心思想。例如，书的抽象就是书的简介，与之相对的是整本书。

当设计一个类时，不应该考虑任一个方法的实现，即不应该担心类的方法如何达成它的目标。将规范说明与实现分开，能让你专心于更少的细节，所以能让你的工作更容易，出错概率更低。详细的设计良好的规范说明，有助于让实现更易成功。

P.4 **注：**抽象的过程要求你关注什么而不是如何。

如果正确，封装将类定义分为两部分——**接口** (interface) 和 **实现** (implementation)。接口描述程序员使用这个类时必须要了解的一切事情。它包括类的公有方法的方法头，告诉程序员如何使用这些公有方法的注释，及类中公有定义的任何常量。接口部分应该是在你的程序中使用这个类时只需要了解接口。注意，使用某个类的程序称为类的客户 (client)。

实现部分由所有的数据域及所有方法的定义组成，包括公有、私有及保护的。虽然执行客户程序时需要实现，但编写客户程序时应该不需要知道任何实现细节。图 P-2 说明了一个类的封装实现及客户接口。虽然实现对客户是隐藏的，但接口却是可见的，且为客户提供了与实现进行交互的规范机制。

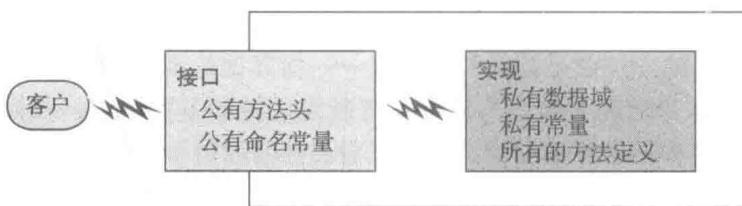


图 P-2 接口在隐藏的实现与客户之间提供了规范的交互机制

接口和实现在 Java 类的定义中是不分开的，它们合在一起。不过你可以随同你的类创建一个独立的 Java 接口。本序言后半部分的内容介绍如何写这样的接口，本书中还会再写几个。



学习问题 1 接口如何区别于类的实现？

学习问题 2 考虑一个不同于汽车的例子，用来说明封装。例子中的哪些部分对应于接口，哪些部分对应于实现？

规范说明方法

将类的目标及方法与其实现分开，对一个成功的软件项目来说至关重要。应该规范说明每个类及方法，而不关心它的实现。写这些描述能捕捉到你最初的想法，并开发它们，从而才能足够明确地实现它们。你写的描述应该作为注释放到程序中有用的地方。你不能在写了程序之后为了应付老师或老板才写注释。

注释

现在来看看为类的方法而写的注释。虽然各企业有自己的注释风格，但 Java 开发者有特定的应该遵从的注释风格。如果程序中含有这种风格的注释，则可以执行一个称为 javadoc 的实用程序，从而得到描述类的文档。这个文档告诉未来使用这些类的人们必须要了解的东西，但忽略了所有实现细节，包括所有的方法定义体。

程序 javadoc 提取类头、所有公有方法的头，及以特定形式写的注释。每个这样的注释必须出现在公有类定义或公有方法头的前面，且必须以 `/**` 开头，以 `*/` 结尾。注释中以符号 `@` 开头的特定的标签（tag）标识方法的不同方面。例如，使用 `@param` 标识参数，`@return` 标识返回值，而 `@throws` 表示方法抛出的异常。本序言中，在注释中会看到几个这样的标签示例。附录 A 详述了如何书写 javadoc 注释。

现在不再进一步讨论 javadoc 注释的规则，而讨论规范说明一个方法的重要方面。首先，你需要写一个简洁的语句来阐述方法的目的或任务。这个语句以动词开头，能让你避免冗长的文字，而那些文字真的是不需要的。

在思考方法的目的时，应该考虑它的输入参数，如果有，要描述它们。还需要描述方法的结果。是让它返回一个值、让它做些动作，还是让它改变参数的状态？在写这些描述时，应该时刻牢记以下理念。

前置条件和后置条件

前置条件（precondition）是一条条件语句，它在方法执行前必须为真。除非前置条件满足，否则不应该使用方法，也不能期待方法正确执行。前置条件可以与方法参数的描述相关。例如，计算 x 平方根的方法可以用 $x \geq 0$ 作为前置条件。

后置条件（postcondition）是一条语句，当前置条件满足且完全执行方法后，它为真。对于一个值方法，后置条件将描述方法返回的值。对于一个 void 方法，后置条件描述所做的动作及对调用对象的任何修改。一般地，后置条件描述方法调用产生的所有影响。

考虑后置条件有助于弄清楚方法的目的。注意到，从前置条件到后置条件没有提到如何做，即我们将方法的规范说明与它的实现分离。

 **程序设计技巧：**不能满足后置条件的方法，即使符合前置条件，也可以抛出异常。（关于异常的讨论见 Java 插曲 2 和 Java 插曲 3。）

职责。前置条件的职责是保证必须满足特定条件。如果是信任的客户，比如同一个类中的另一个方法，负责在调用方法之前满足条件，则该方法不需要检查条件。另一方面，如果该方法负责满足条件，则客户不检查它们。明确声明谁必须检查一组给定条件，既提高了检查的概率，又避免了重复劳动。

P.5

P.6

P.7

P.8

例如，要规范说明前一段提到的求平方根方法，可以在方法头前面写如下的注释：

```
/** Computes the square root of a number.  
 * @param x A real number >= 0.  
 * @return The square root of x.  
 */
```

虽然我们在这个注释中将前置条件和后置条件一起写了出来，但是可以分别指明它们。更安全的技术是让方法承担检查参数的职责。此例中，注释应该如下：

```
/** Computes the square root of a number.  
 * @param x A real number.  
 * @return The square root of x if x >= 0.  
 * @throws ArithmeticException if x < 0.  $\Theta$   
 */
```

程序设计技巧：在方法头之前的注释中充分说明每个公有方法。对于确保方法能正确执行而必须满足的条件，要说明是由方法还是由客户来负责进行检查。以这种方式，既做了检查又不会重复检查。但在调试过程中，方法应该检查前置条件是否满足。

P.9

当使用继承和多态来重写超类中的方法时，子类中的方法可能会出现与超类中的方法不一致的问题。前置条件和后置条件可以帮助程序员避免这个问题。后置条件必须适用于子类中方法的所有版本。重写的方法可以添加到后置条件中——即它能做得更多——但不能做得更少。不过重写的方法不能增加其前置条件。换句话说，它不能比基类中的方法要求得更多。



学习问题 3 假定类 Square 有一个数据域 side 及设置 side 值的方法 setSide。这个方法的方法头和注释是什么？回答这个问题的时候要牢记前置条件和后置条件。

断言

P.10

断言（assertion）是一条关于程序逻辑的某些方法的事实语句。可以将它看作值为真的布尔表达式，或至少在某些点应该为真。例如，前置条件和后置条件是方法开始前及结束后关于条件的断言。如果有一个断言为假，则程序一定有错。

可以将断言作为注释放在代码中。例如，如果在方法定义的某些地方，你知道变量 sum 应该是正的，则可以写如下的注释：

```
// Assertion: sum > 0
```

这样的注释用来说明并不太明晰的某些逻辑。另外，断言为你指明调试期间需要精确检查的代码位置。



学习问题 4 假定你有一个正整数数组。下列语句查找数组中的最大整数。下列循环中的 if 语句之后，应该写一个什么样的断言来当作注释？

```
int max = 0;  
for (int index = 0; index < array.length; index++)  
{  
    if (array[index] > max)  
        max = array[index];  
    // Assertion:  
} // end for
```

Θ Java 插曲 2 讨论异常。

断言语句 (assert statement)。Java 不仅仅能让你写个注释当断言，还能使用 assert 语句强制执行断言，如

```
assert sum > 0;
```

如果保留字 assert 后面的布尔表达式为真，则语句什么也不做。如果它为假，则发生断言错误 (assertion error)，程序中断执行。显示如下的错误信息：

```
Exception in thread "main" java.lang.AssertionError
```

可以在 assert 语句后添加第二个表达式来进一步说明这条错误信息。第二个表达式必须表示一个值，而在错误信息中它是作为字符串显示的。例如，语句

```
assert sum > 0 : sum;
```

当 sum ≤ 0 时在错误信息中添加了 sum 的值。比如，错误信息可能是

```
Exception in thread "main" java.lang.AssertionError: -5
```

默认情况下，程序执行时是禁用 assert 语句的。所以程序完成后可以将 assert 语句留在程序中，而不会浪费运行时间。当执行一个程序时，如果想让它们执行，就必须要启用 assert 语句。如何启用它们依赖于编程环境。^①

 **注：**程序中的断言明示出必须为真的逻辑。在 Java 中，可以使用一条 assert 语句写一个断言。它的格式如下：

```
assert boolean_expression : valued_expression;
```

如果第一个表达式为假，则可选的第二个表达式的值将出现在错误信息中。

 **程序设计技巧：**使用 assert 语句是发现程序逻辑错误的简单有效的方法。除了可用于这个目的之外，留在程序中的断言还能向修改或扩展程序的人阐明你的逻辑。记住，Java 会忽略 assert 语句，除非程序的使用者指定了其他的选项。

 **程序设计技巧：**调试时使用 assert 语句，能使方法强制满足前置条件。但是 assert 语句不能替代 if 语句。应该将 assert 语句作为程序设计的辅助手段，而不是程序逻辑的一部分。

Java 接口

本序言前面的内容中，我们提到接口这个术语，当在你的程序中要使用某个类时，由接口告诉你必须要知道的所有东西。虽然一个 Java 类与实现它的接口合在一起，但是也可以写一个单独的接口。

Java 接口 (Java interface) 是一个程序组件，它声明了一些公有方法，且能定义公有命名常量。这样的接口应该含有说明方法的注释，以便为实现它们的程序员提供必要的信息。有些接口描述了类中的所有公有方法，还有一些仅说明特定的方法。

当写一个类来定义接口中声明的方法时，称这个类实现 (implement) 了接口。实现接口

^① 如果使用 Oracle 的 Java Development Kit(JDK)，则命令 `java -ea MyProgram` 可使 MyProgram 启用断言。有关断言的更详细的内容，可在互联网上使用“Java assertions”(Java 断言) 进行查找。

的类必须定义接口中说明的每个方法的方法体。但是接口可能没有声明类中定义的每个方法。

你能写自己的接口，也能使用 Java 类库中定义的接口。当写一个 Java 接口时，可以将它放在它自己的文件中，即接口和实现接口的类在两个独立的文件中。

写一个接口

P.13 Java 接口的开头很像是类的定义，不过要用保留字 `interface` 替代 `class`，即接口的开头是如下语句：

```
public interface interface-name
```

而不是

```
public class class-name
```

接口可以含有任意多个公有方法头，每个方法头的后面是一个分号。接口不声明类的构造方法，也不能声明静态或终极方法。注意，接口中的方法默认是公有的，故在方法头中可以省略 `public`。接口还可以定义任意多个公有命名常量。

P.14  示例。想象如圆、正方形或一块地这样的对象，它们既有周长又有面积。假定我们想让这种对象的类有一个返回数量值的访问方法。如果实现这些类的程序员不是同一个人，则他们可能会用不同的方式来说明这些方法。为确保定义这些方法的类有统一的格式，我们可以写一个接口，如程序清单 P-1 所示。这个接口为程序员提供了方法说明的简单概要。程序员应该不必去查看实现它们的类就能使用这些方法。

程序清单 P-1 接口 Measurable

```
1  /** An interface for methods that return
2   * the perimeter and area of an object.
3   */
4  public interface Measurable
5  {
6      /** Gets the perimeter.
7      * @return The perimeter. */
8      public double getPerimeter();
9
10     /** Gets the area.
11     * @return The area. */
12     public double getArea();
13 } // end Measurable
```

将接口定义保存在一个与接口名同名的文件中，后面加上 `.java`。例如，前面的接口在文件 `Measurable.java` 中。

 程序设计技巧：Java 接口是写注释的好地方，是用来说明每个方法的目的、参数、前置条件及后置条件的地方。用这种方式，可以在一个文件中说明一个类，而在另一个文件中实现它。

 注：接口可以声明数据域，但它们必须是公有的。通常，类的数据域是私有的，故接口中的任何数据域表示的都应该是命名常量。所以它们应该是公有的、终极的及静态的。

注：接口中声明的方法不能是静态的，也不能是终极的。但是可以在实现接口的类中声明这样的方法。

示例。假定你最终想定义人名的类。最开始或许是定义如程序清单 P-2 所示的 Java 接口，为这个人名类说明方法。限于篇幅，我们只为最开始的两个方法添加了注释。这个接口提供了整个类中所需方法的规范说明。当实现附录 B 中程序清单 B-1 所示的如 Name 这样的类时可以使用它。另外，只看这个接口，就应该能为类写一个客户。

程序清单 P-2 接口 NameInterface

```

1  /** An interface for a class of names. */
2  public interface NameInterface
3  {
4      /** Sets the first and last names.
5          @param firstName A string that is the desired first name.
6          @param lastName  A string that is the desired last name. */
7      public void setName(String firstName, String lastName);
8
9      /** Gets the full name.
10         @return A string containing the first and last names. */
11     public String getName();
12
13    public void setFirst(String firstName);
14    public String getFirst();
15
16    public void setLast(String lastName);
17    public String getLast();
18
19    public void giveLastNameTo(NameInterface aName);
20
21    public String toString();
22 } // end NameInterface

```

注意，方法 giveLastNameTo 的参数的数据类型是接口——NameInterface——而不是像 Name 这样的类。我们将在段 P.19 的开头来谈论接口作为数据类型的话题。现在，只需知道接口不应该限制实现它的类的名字。

注：命名一个接口

接口名，特别是 Java 中标准的接口名，常常以“able”结尾，例如 Measurable。这样的结尾并不总能提供一个好名字，所以也常会使用“er”或是“Interface”作为结尾。与 Java 的异常以“Exception”为结尾一样，接口常以“Interface”为结尾。

实现一个接口

实现接口的任何类，必须要在类定义的开头使用 implements 子句进行说明。例如，如果类 Circle 实现了接口 Measurable，它的开头就是下面这种形式的：

```
public class Circle implements Measurable
```

然后，类必须定义接口中声明的每个方法。本例中，类 Circle 必须至少实现方法 getPerimeter 和 getArea。

如果写一个实现 Measurable 的类 Square，这个类的开头应该是这样的：

```
public class Square implements Measurable
```

且它至少应该定义方法 `getPerimeter` 和 `getArea`。显然，这两个方法的定义不同于它们在类 `Circle` 中的定义。

图 P-3 展示的是 `Measurable`、`Circle`、`Square` 及它们的客户所在的文件。

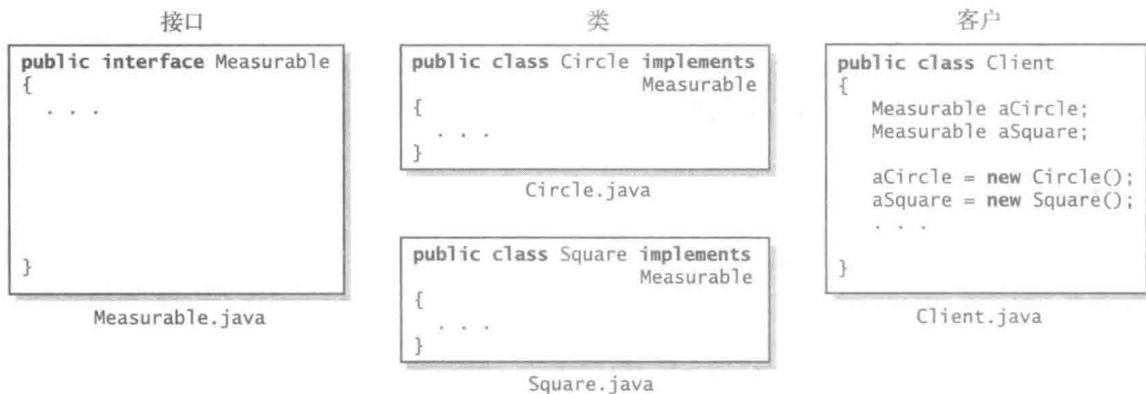


图 P-3 用于接口、实现接口的类及其客户的文件



注：写接口是类的设计人员向其他程序员规范说明方法的一种方式。实现接口是程序员确保类已经定义了某些方法的一种方式。



注：不同的类或许以不同的方式实现同一个接口。例如，可以有多个类实现接口 `Measurable`，且为方法 `getPerimeter` 和 `getArea` 写各自的版本。

P.17



示例。想象用于如圆、球体和圆柱体等不同几何形状的类。其中的每一个几何体都有一个半径。我们可以定义下列接口，让类来实现它：

```
public interface Circular
{
    public void setRadius(double newRadius);
    public double getRadius();
} // end Circular
```

接口能知道已经定义了半径，所以为这个量声明设置方法和获取方法。但是，不能为半径声明数据域。实现接口的类来做这件事。

实现这个接口的类 `Circle` 如下所示：

```
public class Circle implements Circular
{
    private double radius;
    public void setRadius(double newRadius)
    {
        radius = newRadius;
    } // end setRadius
    public double getRadius()
    {
        return radius;
    } // end getRadius
    public double getArea()
    {
        return Math.PI * radius * radius;
    } // end getArea
} // end Circle
```

类定义了一个私有数据域 `radius`, 且实现了接口 `Circular` 中声明的方法 `setRadius` 和 `getRadius`。接口中不能含有像 `radius` 这样的数据域, 因为它是私有的。

 注: 类中定义的方法个数可以超出它实现的接口中声明的方法个数。例如, 类 `Circle` 定义了方法 `getArea`, 它没有包含在接口 `Circular` 中。

多个接口。类可以实现多个接口。如果想这样做, 只需列出所有的接口名, 并以逗号分隔即可。如果类从另一个类派生而来, 则 `implements` 子句永远在 `extends` 子句的后面。所以可以写:

```
public class Circle extends Shape implements Measurable, Circular
```

要想记住这个次序, 只需记着保留字 `extends` 和 `implements` 在类头中以字母序出现即可。

实现多个接口的类必须定义接口中声明的每个方法。如果类实现的多个接口中出现了相同的方法头, 则类中只需定义一个方法。

不能从多个基类派生一个类。这个限制避免了实现继承时可能出现的冲突。Java 接口中含有方法的规范说明, 但不去实现它们。类可以实现这些规范说明, 而不管它们是出现在一个接口中还是出现在多个接口中。通过允许类实现多个接口这种机制, Java 既实现了多重继承, 又去掉了它可能引起的混乱。



学习问题 5 写一个 Java 接口, 规范定义学生类并声明其中的方法。

学习问题 6 定义一个类, 实现前一个问题中你写的接口。要包含数据域、构造方法及至少一个方法的定义。

接口作为数据类型

当声明变量、数据域或方法的参数时, 可以将 Java 接口用作数据类型。例如, 段 P.15 P.19 中的方法 `giveLastNameTo` 有一个类型为 `NameInterface` 的参数:

```
public void giveLastNameTo(NameInterface aName);
```

传给这个方法的任何实参, 必须是实现 `NameInterface` 的类的对象。

为什么不将 `aName` 的类型声明为一个类 (例如 `Name`) 呢? 我们想让接口独立于实现它的类, 因为实现一个接口的类可以有多个。使用 `NameInterface` 作为参数的类型, 能保证方法的实参具有 `NameInterface` 中声明的所有方法。通常, 如果数据类型是接口, 你能保证方法的参数具有特定的方法, 即接口中声明的那些方法。另一方面, 参数只有那些方法。

要是一个类 `C` 的头不含有 `implements NameInterface` 短语, 仍实现了接口中的方法, 又会如何呢? 那你不能将 `C` 的实例作为参数传给 `giveLastNameTo` 方法。



注: 将接口当作变量的类型, 这意味着, 这个变量可以引用一个对象, 它有一组方法且仅有这组方法。



注: 接口类型是引用类型。

P.20

如下变量声明

```
NameInterface myName;
```

使得 `myName` 成为一个引用变量。现在 `myName` 可以指向实现 `NameInterface` 的任意一个类的任意对象。故如果 `Name` 实现了 `NameInterface`, 且有

```
myName = new Name("Coco", "Puffs");
```

则 `myName.getFirst()` 返回指向字符串 "Coco" 的引用。如果类 `AnotherName` 也实现了 `NameInterface`, 且随后写了语句:

```
myName = new AnotherName("April", "MacIntosh");
```

则 `myName.getFirst()` 返回指向字符串 "April" 的引用。



学习问题 7 为能利用 `NameInterface`, 需要对学习问题 5 中写的接口及实现它的类 `Student` 做哪些修改?

派生一个接口

P.21

一旦有了一个接口, 就可以使用继承机制从它派生另一个接口。事实上, 可以从多个接口派生一个接口, 虽然不能从多个类派生一个类。

当一个接口继承另一个接口时, 它具有所继承接口中的所有方法。所以你可以创建一个接口, 它含有已有接口中的方法, 再加上一些新方法。例如, 考虑宠物的类及下列接口:

```
public interface Nameable
{
    public void setName(String petName);
    public String getName();
} // end Nameable
```

可以继承 `Nameable` 来创建接口 `Callable`:

```
public interface Callable extends Nameable
{
    public void come(String petName);
} // end Callable
```

实现 `Callable` 的类必须实现方法 `come`、`setName` 和 `getName`。

P.22

还可以从多个接口派生一个新接口, 如果愿意, 甚至还可以添加更多的方法。例如, 假定除了前两个接口外还定义了下列接口:

```
public interface Capable
{
    public void hear();
    public void respond();
} // end Capable

public interface Trainable extends Callable, Capable
{
    public void sit();
    public void speak();
    public void lieDown();
} // end Trainable
```

则实现 `Trainable` 的类必须实现方法 `setName`、`getName`、`come`、`hear` 和 `respond`, 以及方法 `sit`、`speak` 和 `lieDown`。



注：Java 接口可以从几个接口派生，虽然不能从多个类派生一个类。



学习问题 8 假定含有方法 `setName` 的类 `Pet`，还没有实现段 P.21 中的接口 `Nameable`。你能不能将 `Pet` 的实例当作有下列方法头的方法的参数？

```
void enterShow(Nameable petName)
```

接口内的命名常量

接口可以含有命名常量，即已初始化且声明为终极变量的公有数据域。如果你想实现几个类共享一组通用的命名常量，则可以将常量定义在一个接口中，让类来实现接口。也可以将常量定义在一个单独的类中而不是一个接口中。本节我们讨论这两种方式。不管选择哪种方式，当前都只有一组常量可用。

假定有几个类必须将尺寸转换为公制。可以将换算因子定义为这些类可共享的常量。我们将常量放到一个接口中。

常量的接口。下列接口定义了三个命名常量：

P23

```
public interface ConstantsInterface
{
    public static final double INCHES_PER_CENTIMETER = 0.39370079;
    public static final double FEET_PER_METER = 3.2808399;
    public static final double MILES_PER_KILOMETER = 0.62137119;
} // end ConstantsInterface
```

任何接口除了声明方法外还可以定义常量，不过上面这个接口只含有常量。

要在一个类中使用这些常量，可以在类定义中写 `implements` 子句。然后在整个类内可按名使用常量。例如，考虑下面这个简单的类：

```
public class Demo implements ConstantsInterface
{
    public static void main(String[] args)
    {
        System.out.println(FEET_PER_METER);
        System.out.println(ConstantsInterface.MILES_PER_KILOMETER);
    } // end main
} // end Demo
```

用接口名来限定常量是可选的。但是，如果同一个命名常量定义在一个类实现的多个接口中，则在类中必须用接口名来限定常量。



程序设计技巧：为了保持一致性，始终用相关类或接口的名称限定常量的名称。

P24

常量的类。为了同样的目的，可以不将常量定义在接口中，而是将其定义在类中：

```
public class Constants
{
    private Constants()
    {
    } // end private default constructor
    public static final double INCHES_PER_CENTIMETER = 0.39370079;
    public static final double FEET_PER_METER = 3.2808399;
    public static final double MILES_PER_KILOMETER = 0.62137119;
} // end Constants
```

因为这些常量中的每一个都有唯一的值，每一个都只有一个拷贝就足够了，所以我们将它们声明为静态的。注意私有构造方法。因为类中提供了一个构造方法，所以 Java 就不提供了。因为我们的构造方法是私有的，所以客户不能创建类的实例。

使用这个类很简单，如下例所示：

```
public class Demo
{
    public static void main(String[] args)
    {
        System.out.println(Constants.FEET_PER_METER);
        System.out.println(Constants.MILES_PER_KILOMETER);
    } // end main
} // end Demo
```

必须在常量名的前面加上类的名字及一个点。这可能是一个优点，读你程序的人立刻会明白常量来自哪里。如果这样做比较麻烦，则可以定义常量的一个本地拷贝，如

```
final double FEET_PER_METER = Constants.FEET_PER_METER;
```

然后用它来替代。



设计决策：常量应该定义在接口中还是类中

程序员对这个问题给出的答案似乎是不一致的。即使在 Java 类库中也含有两种形式的例子。一般地，常量定义应该是在类内的实现细节。接口声明方法，所以属于规范说明范畴，而不是实现范畴。将接口仅用于方法是一个合理的准则。

选择类

到目前为止，我们已经讨论过规范说明类和实现类的话题，描述了要说明或实现的类。如果必须从零开始设计一个应用程序，你该如何选择所需的类呢？本节介绍软件设计人员在选择及设计类时会用到的一些技术。虽然在本书中我们会不断提到这些技术，但我们的目的只是向你介绍这些思想，未来的课程会更深入地介绍选择和设计类的方法。

P.25

假定我们正在设计一个你学校使用的注册系统。应该从何处着手？有效的切入点应该是从功能的角度来看待系统，包括：

- 谁来使用系统？与系统交互的人类用户或软件组件称为角色（actor）。所以第一步是列出可能的角色。对于一个注册系统来说，两个角色可能是学生和注册员。
- 每个角色对系统能做什么？脚本（scenario）是角色与系统之间进行交互的功能描述。例如，学生能添加一门课程。这个基本脚本可以变化，从而引出其他脚本。例如，当学生试图添加已经关闭的课程时会发生什么事情？故第二步是确定脚本。做这件事的一种方式是将“当……时会发生什么”问题补充完整。
- 哪些脚本涉及共同目标？例如，我们刚描述的两个脚本与添加一门课程这个共同目标有关。这样的相关脚本集合称为用例（use case）。故第三步是确定用例。

通过画用例图（use case diagram），能得到正在设计的系统所涉及的用例的总体图。图 P-4 是这个简单的注册系统的用例图。每个角色——学生和注册员——用简笔画人来表示。盒式方框表示注册系统，方框中的椭圆是用例。如果角色和用例之间存在交互，则两者之间用线连接起来。

本例中，有些用例涉及一个角色，另外一些涉及两个角色。例如，申请入学只用于学生，注册学生只用于注册员。不过，学生和注册员都能在学生课表中添加一门课程。

 注：用例从角色的角度描述系统，它们不一定就暗示着系统内的类。

标识类

虽然画用例图是朝正确方向前进的一步，但它没有标识出系统内必需的类。这可能会涉及几项技术，而你或许会用到其中的一些。

一项简单的技术是描述系统，然后标识出描述中的名词和动词。名词可能暗示着类，而动词可能暗示着类内相应的方法。鉴于自然语言不严谨，这项技术并不是万无一失的，但它很有用。

例如，可以用一系列步骤来描述图 P-4 中的每个用例。图 P-5 给出的用例描述，是从学生角度添加一门课程这个用例。注意，当系统没有识别出学生或所需的课程已关闭时，分别使用替代的步骤 2a 和步骤 4a。

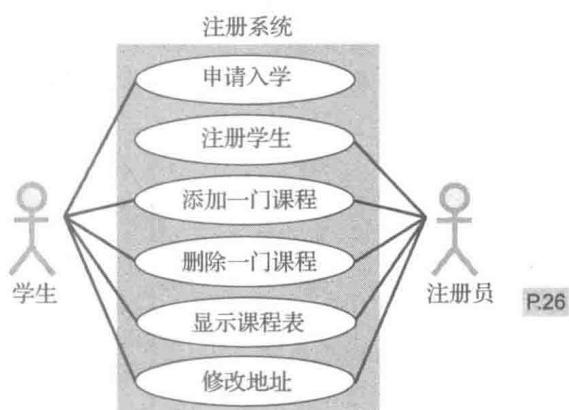


图 P-4 注册系统的用例图

| |
|------------------------------|
| 系统：注册 |
| 用例：添加一门课程 |
| 角色：学生 |
| 步骤： |
| 1. 学生输入身份数据。 |
| 2. 系统确认注册资格。 |
| a. 如果注册资格不合格，要求学生再次输入身份数据。 |
| 3. 学生从课程设置列表中选择课程的具体部分。 |
| 4. 系统确认课程的有效性。 |
| a. 如果课程已经关闭，则允许学生返回步骤 3 或退出。 |
| 5. 系统将课程添加到学生课程表中。 |
| 6. 系统显示修改后的学生课程表。 |

图 P-5 添加一门课程用例的描述

这个描述能暗示出哪些类呢？查看名词，我们能确定一些类，用来表示一名学生、一门课程、所有课程设置列表及学生课程表。动词暗示着一些动作，包括确认学生注册资格是否合格、查看一门课程是否已经关闭，以及将一门课程添加到学生课程表中。下节将介绍的 CRC 卡，是将这些动作转为类的一种方法。

CRC 卡

索引卡是研究类的目标的一项简单技术。每个卡表示一个类。为类选择一个描述性的名字，并将它写到卡的最上面，这是第一步。然后列出表示类的职责（responsibility）的动作。对系统内的每个类都这样处理。最后，标识出类间的交互，即协作（collaboration）。也就是说，在每个类的卡上写出与它有某种交互的其他类的名字。由于卡上的内容，因此将其称为

类职责协作 (Class-Responsibility-Collaboration, CRC) 卡。

例如, 图 P-6 所展示的 CRC 卡表示的是学生已经注册的课程的类 CourseSchedule。注意, 每张卡很小, 所以只能写简单的说明。职责个数必须少, 它暗示着你站在高层考虑很小的类。卡片的尺寸能让你将其放在桌面上, 当你查找协作时可以方便地移动它们。



图 P-6 类职责协作 (CRC) 卡

 学习问题 9 为附录 C 中所给的类 Student 写 CRC 卡。

统一建模语言

P28 图 P-4 中的用例图是更强大表示法的一部分, 这种表示法称为统一建模语言 (Unified Modeling Language, UML)。设计人员使用 UML 来说明软件系统中必需的类及它们的关系。UML 能给出复杂系统的整体视图, 比用自然语言或程序设计语言描述更有效。例如, 英语可能有二义性, Java 代码提供更多的细节。给出明确的类之间的交互图, 是 UML 的强项之一。

除了用例图之外, UML 还能提供类图, 将每个类的描述放在类似于 CRC 卡的方框中。方框内包含类名、属性 (attribute)(数据域) 和操作 (operation)(方法)。例如, 图 P-7 所示为类 CourseSchedule 的方框。一般地, 方框中省略如构造方法、获取方法和设置方法这样的公共操作。

随着设计的推进, 当你描述一个类时可以提供更多的细节。表示域或方法的可见性时, 可以在其名字前加上一个符号, + 用于公有的, - 用于私有的, 而 # 用于保护的。还可以在域、参数或返回值后加上一个冒号, 然后写上它的数据类型。故在图 P-7 中可以有如下的数据域:

```
-courseCount: integer
-courseList: List
```

而方法如下:

```
+addCourse(course: Course): void
+removeCourse(course: Course): void
```

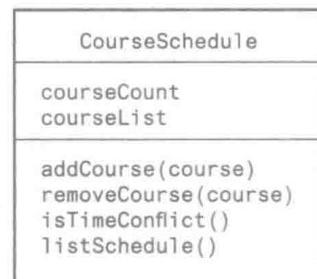


图 P-7 组成类图的类的表示

```
+isTimeConflict(): boolean
+listSchedule(): void
```

在 UML 中表示接口非常类似于表示类的方式，只是要在名字前加上 `<<interface>>`。段 P.14 中接口 `Measurable` 的表示如图 P-8 所示。

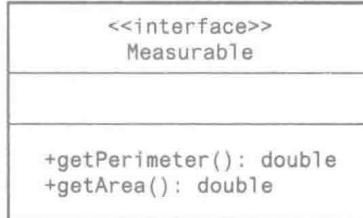


图 P-8 接口 `Measurable` 的 UML 表示



学习问题 10 附录 B 中给出的类 Name 的 UML 类图是什么样的？

在类图中，连接了类的方框的线表示类间的关系，包括继承层次。例如，图 P-9 中的类图表示类 `UndergradStudent` 和 `GradStudent` 都继承自 `Student` 类。空心箭头指向超类。在 UML 中，超类 `Student` 称为 `UndergradStudent` 和 `GradStudent` 的泛化（generalization）。 P.29

如果类实现了一个接口，则从类到接口间画一条带空心箭头的虚线。例如，段 P.16 中的类 `Circle` 实现了接口 `Measurable`。图 P-10 展示了这个关系。

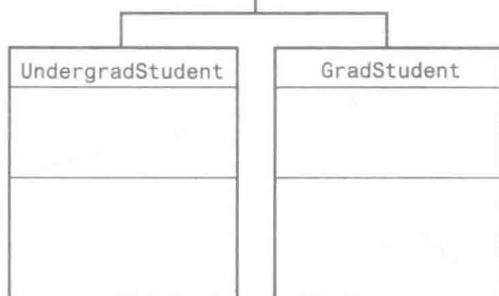
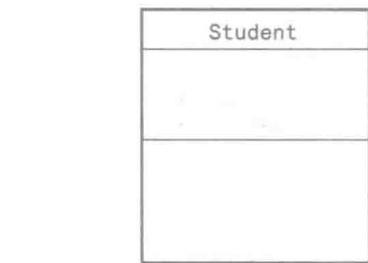


图 P-9 表示基类 `Student` 及其两个子类的类图

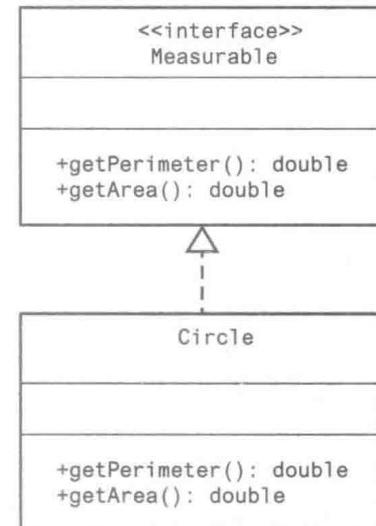


图 P-10 展示实现了接口 `Measurable` 的类 `Circle` 的类图

关联（association）是两个不同类的对象间的关系。基本上，关联就是 CRC 卡称为协作的关系。例如，类 `Student`、`CourseSchedule` 和 `Course` 之间存在的关系。图 P-11 展示了 UML 如何用箭头表示这些关系。例如，类 `CourseSchedule` 和 `Course` 之间的箭头，表

示类 `CourseSchedule` 的对象和类 `Course` 的对象之间的关系。这个箭头指向 `Course`，表示职责。所以，`CourseSchedule` 对象应该能告诉我们它包含的课程，但 `Course` 对象不能告诉我们它属于哪个课程表。UML 称这种表示为可操纵性 (navigability)。

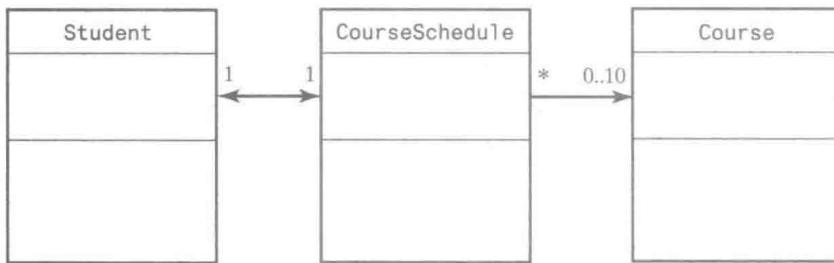


图 P-11 有关联关系的部分 UML 类图

这种特殊的关联称为单向的 (unidirectional)，因为它的箭头指向一个方向。两端都有箭头的线表示的关联称为双向的 (bidirectional)。例如，`Student` 对象能找到它的课程表，而 `CourseSchedule` 对象能找到它所属的学生。可以假定，不带箭头的线表示的关联的可操纵性，在设计的当前阶段尚未确定。

每个箭头的末端都是数字。在 `CourseSchedule` 和 `Course` 之间箭头的前端，如你所见标注的是 0..10，这个符号表示每个 `CourseSchedule` 对象与 0 ~ 10 门课程关联。这个箭头的另一端是星号，它的含义与记号 0..∞ 是一样的。每个 `Course` 对象可以与许许多多的课程表相关联——或者一个都不关联。类图还表示了 `Student` 对象和 `CourseSchedule` 对象之间的关系。箭头两端的记号称为关联的基数 (cardinality) 或多样性 (multiplicity)。



学习问题 11 将图 P-9 和图 P-11 合成一个类图。然后添加类 `AllCourses`，它表示本学期开设的所有课程。你必须添加的新关联有哪些？

重用类

P.30

当你首次着手写程序时，很容易有这样的印象，即每个程序都是从零开始设计且编写的。相反，大多数软件是融合了已有组件与新组件而形成的。这种机制节省了时间和经费。另外，已有的组件已经用过很多次了，所以更易测试且更可靠。

例如，公路模拟程序可能包含一个新的公路对象来模拟新的公路设计，但它或许使用已在其他程序中设计的汽车类来模拟汽车。当你标识出项目中所需的类时，应该看看这些类是否已存在。能不能使用它们，或是把它们当作新类的基类？

P.31

当设计新类时，应该设法保证它们在未来容易重用。必须明确标出类的对象如何与其他的对象进行交互。这是我们在本序言的第一段讨论过的封装原则。但封装不是唯一要遵守的原则。设计类时还必须让对象通用，而不是专为某个程序量身定做。例如，如果程序要求所有的模拟汽车只向前移动，你也应该让汽车类包含后退动作，其他的一些模拟可能会要求汽车后退。

不可否认，你无法预知你的类在未来的所有用途，但你可以而且也应该避免这种依赖性，以免限制其日后的使用。第 18 章介绍了设计类时应始终将其未来的使用牢记在心。

利用本序言讨论的原则来设计一个带接口、可重用，且有适用于 javadoc 注释的类，

是要花工夫的。设计一个满足具体问题的方案花的时间会少一些。当你或其他程序员需要重用接口或类时，付出终有回报。如果你写组件时考虑了未来，它们的每次使用就会更快、更容易。从长远来看，真正的软件开发人员运用这些原则是省时的，因为节省时间即节省了金钱，所以你应该运用它们。

练习

1. 考虑段 P.15 中定义的接口 `NameInterface`。我们只为两个方法写了注释。为另外的每个方法写出符合 `javadoc` 风格的注释。
2. 考虑段 P.16 和段 P.17 给出的类 `Circle` 和接口 `Circular`。
 - a. 是客户还是方法 `setRadius` 负责保证圆的半径是正数？
 - b. 为方法 `setRadius` 写一个前置条件和一个后置条件。
 - c. 为方法 `setRadius` 写适合 `javadoc` 风格的注释。
 - d. 修改方法 `setRadius` 及它的前置条件和后置条件，改变回答问题 a 时提到的职责。
3. 为称为 `Counter` 的类写一个 CRC 卡及类图。这个类的对象用于统计，所以它将记录次数，这是一个非负整数。包括的方法有：为给定整数设置计数器，计数器加 1，计数器减 1。另外，还有将当前计数器作为整数返回的方法，将当前计数器作为可显示在屏幕上的字符串返回的方法 `toString`，测试当前计数器是否为 0 的方法。
4. 假定想为餐馆设计软件，给出下单及结账的用例，列出可能的类的列表。挑选其中的两个类，为它们写 CRC 卡。

项目

1. 为练习 3 设计的类 `Counter` 创建一个接口，包括能用于 `javadoc`、说明类中方法的注释。所有的方法都不允许计数器的值为负数。
2. a. 为附录 C 的程序清单 C-3 中给出的 `CollegeStudent` 类写一个 Java 接口。
b. 类 `CollegeStudent` 实现问题 a 中所定义的接口，要做哪些修改？
3. 假定你想设计一个类，每次给它一个数。类中计算到目前为止所给数的最小值、次小值及平均值。为这个类创建一个接口，包括说明类中方法的符合 `javadoc` 风格的注释。
4. 考虑分数类 `Fraction`。每个分数都有符号，且有整数的分子和分母。你的类应该能对两个分数进行加法、减法、乘法和除法运算。这些方法应该有一个分数作为参数，且应该将操作结果作为分数返回。类还应该能查找分数的倒数、比较两个分数、确定两个分数是否相等，及将分数转换为字符串等。

这个类应该能处理分母为零的情况。分数总应该表示为最简的形式，且类应该负责检查这个条件。例如，如果用户试图创建一个如 $4/8$ 这样的分数，则类应该将分数设置为 $1/2$ 。同样，所有算术运算的结果也应该是最简的形式。注意，一个分数可能是不正确的——分子大于分母，这样的分数应该表示为最简的形式。

设计但不实现类 `Fraction`。从为这个类写 CRC 卡入手。然后写一个 Java 接口，声明每个公有方法，包括说明每个方法的 `javadoc` 风格的注释。

5. 写一个 Java 类 `Fraction`，实现前一个项目中设计的接口。从写合理的构造方法入手。设计并实现有用的私有方法，包括说明它们的注释。

为将像 $4/8$ 这样的分数化为最简形式，需要将分子和分母同除以它们的最大公约数。 4 和 8 的最大公约数是 4 ，所以当将 $4/8$ 的分子和分母同除以 4 时，得到分数 $1/2$ 。下列递归算法找到两个正整数的最大公约数：

```

Algorithm gcd(integerOne, integerTwo)
while (integerTwo != 0)
{
    r = integerOne % integerTwo
    integerOne = integerTwo
    integerTwo = r
}
return integerOne

Algorithm gcd(integerOne, integerTwo)
if (integerOne % integerTwo == 0)
    result = integerTwo
else
    result = gcd(integerTwo, integerOne % integerTwo)
return result

```

如果强制让分数的分母为正数，则很容易确定分数的正确符号。但是你的实现必须处理客户可能提供的负数分母的情况。

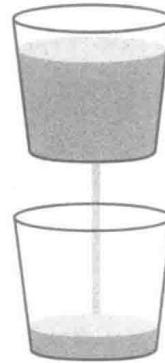
写一个充分展示你的类的程序。

6. 混合数含有整数部分和分数部分。使用前一个项目中设计的类 **Fraction**，为混合数设计类 **MixedNumber**。为 **MixedNumber** 提供类似于 **Fraction** 的操作，即为混合数提供设置、获取、加法、减法、乘法及除法操作。混合数的分数部分应该是最简形式，分子应严格小于分母。为这个类写一个 Java 接口，包括 **javadoc** 注释。
7. 实现前一个项目设计的类 **MixedNumber**。尽可能使用 **Fraction** 中的操作。例如，要让两个混合数相加，将它们转为分数，使用 **Fraction** 类的加法操作进行相加，然后将结果分数转为混合数。其他的算术操作使用类似的技术实现。

如果不仔细，那么混合数的符号可能是个难处理的问题。数学上规定，整数部分的符号与分数部分的符号一致是有意义的。例如，如果有一个负数分数，混合数的 **toString** 方法将得到字符串 "**-5 -1/2**"，而不是 "**-5 1/2**"，这是通常所期望的。下面的解决方案可能会大大简化计算。

混合数的符号表示为字符数据域。一旦设置了这个符号，则让整数和分数部分都为正。当创建混合数时，如果给定的整数部分非零，则让整数部分的符号与混合数的符号一样，而忽略分数的分子和分母的符号。但是，如果给定的整数部分为零，则让所给分数的符号与混合数的符号一致。

8. 考虑两个相同的桶。一个桶挂在天花板的钩子上，且装有液体。另一个桶是空的，且放在地板上，正对在第一个桶的下面。突然在第一个桶的底部有一个小洞，液体从满桶流向地板上的空桶中，如下图所示：



液体不断地流出，直到上面的桶为空时为止。

为说明这个动作的程序设计类。当程序开始执行时，它应该显示泄漏发生前两个桶的原始条件。判断泄漏是自然发生还是用户给出信号时，例如按下回车键或是按下鼠标。如果是后者，应该让用户将光标放在桶底，用来指出泄漏发生的位置。

写 CRC 卡及 Java 接口，包括 javadoc 风格的注释。

9. 实现前一个项目中泄漏桶程序的设计。
10. 里程表记录汽车的行驶里程。机械式里程表含有几个转盘，当车行驶时它们会转动。每个转盘显示 0 ~ 9 的一位数字。最右侧的转盘转得最快，每行驶 1 英里[⊖]其数加 1。一旦转盘转到 9 后，下一英里时它又转回 0，而它左侧转盘的值增 1。可以推广这种转盘的行为，即赋给它们的符号不仅仅是 0 ~ 9。这种转盘计数器的示例包括：
 - 一个二进制里程表，其每个转盘显示 0 或 1。
 - 有三个转盘的桌面日期显示表，分别用于显示年、月和日。
 - 掷骰子显示器，每个转盘显示一个骰子上的点数。
 写一个用于通用转盘计数器的 Java 接口，最多有 4 个转盘。另外，写一个 Java 接口，用于表示转盘的类。包括 javadoc 风格的注释。
11. 实现前一个项目中描述的通用转盘计数器的设计。写程序计算 4 个骰子所显示的值的和大于 12 的概率。(和大于 12 的骰子数，除以可能的骰子总数。) 使用转盘计数器实例得到所有可能的 4 个六面骰子。例如，如果转盘从 [1, 1, 1, 1] 开始，则转盘计数器将如下增加：[1, 1, 1, 2]、[1, 1, 1, 3]、[1, 1, 1, 4]、[1, 1, 1, 5]、[1, 1, 1, 6]、[1, 1, 2, 1]，等等。
12. 使用前两个项目描述的通用转盘计数器的设计和实现，写一个类来表示有 4 个转盘的桌面日期显示器，每个转盘分别表示星期、月、日和年。注意，一天的星期名和日期数的变化速度是一样的，但它们的基点数是不一样的。它们不属于同一个转盘计数器。
13. (游戏) 设计并实现视频游戏中的角色类。类的数据应该含有角色的名字、身高、体重、道德和健康。我们把一个角色的道德表示为一个实数，从 -1.0 (邪恶) 到 0.0 (中性) 再到 1.0 (善良)。角色的健康表示为一个实数，从 0.0 (死亡) 到 1.0 (完全健康) 之间。初始时，角色是完全健康的，且有中性的道德。客户应该能初始化角色的名字、身高和体重。包括下列操作：
 - `heal`——按照客户提供的百分比来增大角色的健康值。
 - `injure`——按照客户提供的百分比来减小角色的健康值。
 - `change`——基于用户提供的参数按照随机百分比改变角色的道德值。正的参数增大道德值，负的参数减小道德值。
 - `toString`——返回描述角色数据域当前值的一个字符串。
 为规范说明的每个方法写 javadoc 风格的注释。
14. (财务) 像支票账户、信用卡账户和贷款账户这样的金融账户有一些共同点。新交易——借和贷——可以记入账户中，且如果它们还没有被接受，则可以被取消。账户可以被接受，并可以提供其当前余额和所有交易。
 - a. 设计类 `Transaction` 和类 `FinancialAccount`。为规范说明的每个方法提供 javadoc 风格的注释。
 - b. 画 UML 类图，包括类 `Transaction`、`FinancialAccount`、`CreditCardAccount` 和 `CheckingAccount`。
15. (电子商务) 设计并实现在线商店中可用产品的类。每个产品有名字、类别、描述、商店 ID、制造商 ID、价格和当前库存。初始时，产品必须有用户指定的名字、商店 ID 和价格。包括的方法有基于用户指定的值设置产品的属性，还要提供两个方法：调整产品的当前库存以及返回描述产品数据域当前值的一个字符串。为规范说明的每个方法提供 javadoc 风格的注释。

[⊖] 1 英里 = 1609.344 米。——编辑注

包

先修章节：序言、附录 C

目标

学习完本章后，应该能够

- 描述抽象数据类型 (ADT) 的概念
- 描述 ADT 包
- 在 Java 程序中使用 ADT 包

本章基于序言中提出的封装和数据抽象的概念，提出了抽象数据类型的表示。或许你已经知道，像 `int` 或 `double` 这样的数据类型 (data type) 是一组值及使用某种特定的程序语言定义的这些值上的操作。相比之下，抽象数据类型 (Abstract Data Type, ADT) 是在概念层面上定义的一组值及这些值上的操作的规范说明，是独立于任何程序设计语言的。数据结构 (data structure) 是使用一种程序设计语言实现的 ADT。

本章还概括了对象分组的概念。集合 (collection) 是将其他对象组成一组，并为它的客户提供不同服务的 ADT。具体来说，一个典型的集合，能让客户添加、删除、获取及查询它表示的对象。不同的集合用于不同的目的。它们的行为是抽象的，且目的因集合而不同。虽然集合是一个抽象数据类型，但是，ADT 不一定是集合。

我们将规范说明并使用 ADT 包，以便给出集合及抽象数据类型的一个示例。为此，我们将为包提供一个 Java 接口。仅需了解这个接口，就能将包用在 Java 程序中。不需要知道包中的项是如何保存的，也不需要知道包的操作是如何实现的。实际上，你的程序不依赖于这些规范说明。正如你将看到的，程序的这个重要特性就是数据抽象的全部。

什么是包

1.1

设想一个纸袋、可重复使用的布袋或者一个塑料袋子。当人们购物、打包午餐或吃土豆片时会用到袋子。袋子里装着东西。在日常用语中，袋子是一类容器。但在 Java 中，容器 (container) 是一个对象，它的类派生于标准类 `Container`。这样的容器用在图形程序中。在 Java 中，不把包 (bag) 看作一个容器，而看作一种集合。

包与其他集合的区别是什么呢？包仅仅是包含它的项。既不能按某种方式排定项的次序，也不能避免重复的项。大多数的行为可由其他类型的集合执行。当描述本章设计的集合的行为时，要谨记一点，就是我们受一个实际的物理包的启发来规范说明一个抽象的概念。例如，纸袋内装着不同大小和形状的东西，且没有特定的次序，也不考虑它的重复性。我们的抽象包将含有无序且可能重复的对象，但我们强调，这些对象有相同或相关的数据类型。



注：包是没有特定次序的对象的有限集合。这些对象具有相同或相关的数据类型。包可以包含重复项。

包的行为

因为包中含有有限个对象，所以报告它含有多少个对象可能是包的行为之一：

1.2

- 得到包中当前的项的个数。

相关的行为是检测包是否为空：

- 看看包是否为空。

我们应该能添加和删除对象：

1.3

- 将一个给定对象添加到包中。
- 从包中删除一个对象。
- 如果可能，从包中删除一个特定对象。
- 从包中删除所有对象。

虽然你不想让杂货店的打包员将 6 个汤罐头扔到包中的面包和鸡蛋上面，不过添加操作并没有指明对象要放在包中的什么位置。记住，包中的内容无序。另外，三个删除操作中的第一个只是删除它能删除的任何对象。这个操作就像是伸手到袋子里把东西拿出来一样。而第二个删除操作是在包中查找特定的项。如果找到，则拿出它。如果包中含有多个相等的对象都满足你的查找条件，那么删除其中的任意一个。如果在包中找不到对象，那就不能删除它，就是这样。最后一个删除操作只是清空包中的所有对象。

你买了多少个狗食罐头？你记得拿凤尾鱼酱了吗？袋子里有什么？可用下列操作回答这几个问题：

1.4

- 统计包中某个特定对象的个数。
- 测试包中是否含有某个特定对象。
- 查看包中所有的对象。

现在我们的行为足够了。此时，我们将所有的行为写在一张纸上，或写在序言中提出的如图 1-1 所示的类职责协作 (CRC) 卡上。

因为包是一个抽象数据类型，所以我们仅描述它的数据并规范说明它的操作。我们不指明如何保存数据或如何实现它的操作。例如不要考虑数组。首先，你需要清楚地知道包都有哪些操作：注意力关注于什么操作可行，而不是它们如何做的。即在程序中能使用包之前，就需要一组详细的规范说明。实际上，甚至在你还没有确定程序设计语言时，就应该先规范说明包的操作。

1.5

| Bag |
|------------------|
| 职责 |
| 得到包中当前的项数 |
| 查看包是否为空 |
| 将给定对象添加到包中 |
| 从包中删除一个未指定的对象 |
| 如果可能，从包中删除某个特定对象 |
| 从包中删除所有对象 |
| 统计某个对象在包中出现的次数 |
| 测试包是否含有某个特定对象 |
| 查看包中的所有对象 |
| 协作 |
| 包能够含有的对象的类 |

图 1-1 用于类 Bag 的 CRC 卡



注：因为抽象数据类型描述了独立于程序设计语言的数据组织方式，所以实现它时你可以对程序设计语言有所选择。

规范说明一个包

在用 Java 实现包之前，必须描述它的数据，并详细说明对应于包行为的方法。我们将命名方法，选择它们的参数，确定它们的返回值类型，并写出注释充分描述对包数据的影

响。当然我们最终的目的是写出每个方法的 Java 头和注释，首先我们用伪代码来描述方法，然后用统一建模语言（UML）进行表示。

1.6 CRC 卡中的第一个行为引出一个方法，该方法返回当前包中项的个数。对应的方法没有参数，它返回一个整数。使用伪代码，我们有下列的规范说明：

```
// 返回包中当前含有的项的个数
getCurrentSize()
```

可以使用 UML 将方法表示为：

```
+getCurrentSize(): integer
```

并将这一行加到类图中。

可以使用一个布尔值方法来测试包是否为空，同样该方法没有参数。用伪代码及 UML 描述这个方法的规范说明如下：

```
// 如果包为空则返回真
isEmpty()
```

及

```
+isEmpty(): boolean
```

将这一行加到类图中。

 **注：**因为通过查看 `getCurrentSize` 是否返回 0 就能检测包何时为空，所以并不真的需要操作 `isEmpty`。但是它是所谓的便利方法（convenience method），所以很多集合都提供这样一个操作。

1.7 现在想向包中添加给定的对象。可以将这个方法命名为 `add`，且有一个表示新项的参数。可以写出下列伪代码：

```
// 将新项添加到包中
add(newEntry)
```

我们可能想让 `add` 作为一个 `void` 方法，但是如果包满则不能将新项添加到包中。这种情况下我们该如何办呢？



设计决策：当不能添加新项时方法 `add` 将如何处理？

当 `add` 不能完成任务时，我们可以有下面两种选择：

- 什么也不做。我们不能添加另外的项，所以忽略这个项并且不改变包。
- 不改变包，但告诉客户添加是不可能的。

第一个选择是简单的，但会让客户疑惑到底发生了什么。当然，我们可以规定 `add` 的前置条件，即包必须不满。这样客户要负责避免将新项添加到满包中。

第二个选择更好一些，且规范说明或实现时也不太难。我们如何告诉客户添加是否成功？标准 Java 接口 `Collection` 规定，如果添加没有成功则发生异常。稍后我们使用另一种方式完成这个方法。显示一条错误信息并不是好的选择，因为所有的书面输出应该由客户决定。因为添加操作或者成功或者不成功，所以我们可以让方法 `add` 返回一个布尔值。

故，用 UML 规范说明 `add` 方法如下：

```
+add(newEntry: T): boolean
```

其中 T 表示 newEntry 的数据类型。



学习问题 1 假定 aBag 表示一个有有限容量的空包。写伪代码，将用户提供的字符串添加到包中，直到包满。

有 3 个动作涉及从包中删除项：删除所有的项，删除任意一个项，删除某个特定项。假定我们用伪代码为这些方法命名并规范说明其参数，如下所示：

```
//删除包中的所有项  
clear()  
  
//删除包中一个未指定的项  
remove()  
  
//如果可能，从包中删除一个特定项  
remove(anEntry)
```

这些方法的返回类型是什么？

方法 clear 可以是一个 void 方法：我们只想清空一个包，不获取它的任何内容。所以，在 UML 中该方法写为：

```
+clear(): void
```

如果第一个 remove 方法从包中删除一个项，则方法可以简单地返回被删除的对象。它的返回类型则为 T，这是包中项的数据类型。在 UML 中，我们有：

```
+remove(): T
```

现在，对于从空包中删除对象的操作，我们可以用返回 null 做出响应了。

如果包中不含有某个项，则第二个 remove 方法不能从包中删除这个项。可以让方法返回一个布尔值，类似于 add 方法那样，用这个值来表示成功与否。或者，方法可以返回被删对象，或者，如果不能删除这个对象则返回 null。下面用 UML 表示的规范说明，是这个方法的两种可能的版本——我们必须二选一：

```
+remove(anEntry: T): boolean
```

或者

```
+remove(anEntry: T): T
```

如果 anEntry 等于包中的一个项，则这个方法的第一个版本将删除这个项并返回真。即使方法没能返回被删除的项，客户也能有方法的参数 anEntry，它等于被删除的项。故我们选择这个版本，它与标准接口 Collection 是一致的。



学习问题 2 在一个类内同时具有刚描述的 remove(anEntry) 的两个版本合法吗？解释之。

学习问题 3 在一个类内同时具有 remove 的两个版本，一个不带参数而另一个带一个参数，这样合法吗？解释之。

学习问题 4 给定学习问题 1 中创建的满包 aBag，写伪代码语句，删除并显示包中所有的字符串。

1.10

其他动作并不改变包的内容。其中的一个动作是统计包中给定对象的出现次数。我们先用伪代码后用 UML 规范说明它，如下所示：

```
//统计给定项在包中出现的次数
getFrequencyOf(anEntry)
+getFrequencyOf(anEntry: T): integer
```

另一个方法测试包是否含有给定对象。使用伪代码及 UML 给出的规范说明如下：

```
//测试包是否含有给定项
contains(anEntry)
+contains(anEntry: T): boolean
```



学习问题 5 给定学习问题 1 中创建的满包 aBag，写伪代码语句，显示 aBag 中字符串 "Hello" 出现的次数（如果有的话）。

1.11

最后，我们想看看包的内容。不是提供显示包中项的方法，而是定义一个方法来返回保存这些项的数组。这样，客户可以按照自己的意愿显示部分或全部的项。下面是最后这个方法的规范说明：

```
//返回包中所有项的数组
toArray()
+toArray(): T[]
```

当方法返回一个数组时，它通常应该定义一个新的数组来返回。我们还将说明这个方法的细节。

1.12

与为包中的方法提供前面那些规范说明时一样，我们使用 UML 符号来表示它们。图 1-2 中显示了这些结果。

注意，CRC 卡和 UML 并没有反映所有的细节，例如我们在前面的讨论中提到过的假定和特殊情形。但是，在标出了这样的条件后，你应该规范说明每种情形下方法应有的动作。应该写下你的决策，想让方法如何动作，就像我们写在下表中的样子。然后，可以将这些非形式化的描述放到说明方法的 Java 注释中去。

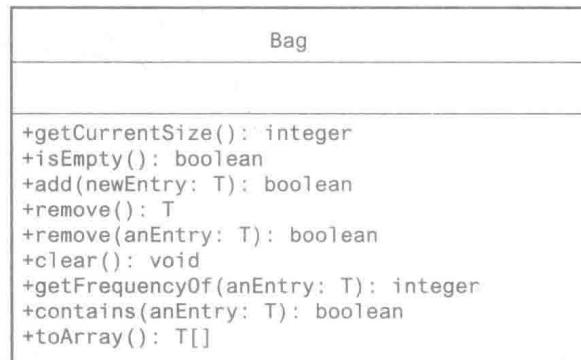


图 1-2 类 Bag 的 UML 表示

| 抽象数据类型：Bag | | | |
|------------|------------------|----------------------------|--|
| 数据 | | | |
| 操作 | 伪代码 | UML | 描述 |
| | getCurrentSize() | +getCurrentSize(): integer | 任务：报告包中当前的对象个数 输入：无 输出：包中当前的对象个数 |

(续)

| 伪代码 | UML | 描述 |
|--------------------------|--------------------------------------|--|
| isEmpty() | +isEmpty(): boolean | 任务：查看包是否为空 输入：无 输出：根据包是否为空返回真或假 |
| add(newEntry) | +add(newEntry: T): boolean | 任务：将给定的对象添加到包中 输入：newEntry 是一个对象 输出：根据添加是否成功返回真或假 |
| remove() | +remove(): T | 任务：如果可能，删除包中一个未指定的项 输入：无 输出：如果删除成功则返回被删除的对象，否则返回 null |
| remove(anEntry) | +remove(anEntry: T): boolean | 任务：如果可能，删除包中一个指定的对象 输入：anEntry 是一个对象 输出：根据删除是否成功返回真或假 |
| clear() | +clear(): void | 任务：从包中删除所有对象 输入：无 输出：无 |
| getFrequencyOf (anEntry) | +getFrequencyOf(anEntry: T): integer | 任务：统计包中一个对象出现的次数 输入：anEntry 是一个对象 输出：包中 anEntry 出现的次数 |
| contains(anEntry) | +contains(anEntry: T): boolean | 任务：测试包是否含有某个特定对象 输入：anEntry 是一个对象 输出：根据 anEntry 是否出现在包中返回真或假 |
| toArray() | +toArray(): T[] | 任务：获取包中所有的对象 输入：无 输出：当前包中项的新数组 |



设计决策：当出现特殊条件时应该怎样办？

作为类的设计者，必须要针对如何处理特殊条件给出相关的决策，并将这些决策包含在规范说明中。ADT 包的文档应该反映前面讨论过的这些决策和细节。

一般可以用几种方式声明特殊情形。你的方法可能采用下列对策：

- 假定无效的情形不会发生。这个假定并不像听起来那么幼稚。方法可以声明一种假设，即前置条件，这是客户必须遵守的限制条件。然后由客户检查在方法调用前这个前置条件是否满足。例如，方法 remove 的前置条件可能是包不能为空。注意到，客户可以使用 ADT 包的其他方法，例如 isEmpty 和 getCurrentSize 来辅助完成这个任务。只要客户遵守限制，无效的情形就不会发生。
- 忽略无效情形。当给了无效数据时，方法可能简单到什么也不做。但是什么都不做会让客户疑惑到底发生了什么。
- 猜测客户的意图。与前一个选择一样，这个选择可能为客户带来麻烦。
- 返回一个表示问题的值。例如，如果客户试图从空包中 remove (删除) 一项时，remove 方法可以返回 null。返回的值必须是不在包中的值。
- 返回一个布尔值，表示操作的成功或失败。
- 抛出一个异常。



注：抛出异常通常是 Java 方法运行期间处理遇到的特殊事件的理想方法。方法可以简单地报告问题而不用决定要做什么。异常能让每个客户根据自己的特殊情形按需处理。Java 插曲 2 将介绍异常的基本机制。



注：ADT 规范说明的初稿常常忽视或忽略你确实需要考虑的情形。你可能为了简化初稿而有意忽略这些。一旦写好了规范说明中的大部分内容，就可以关注这些细节，而让规范说明更完善。

一个接口

1.13

随着规范说明越来越详细，也越发影响到你对程序设计语言的选择。最终，你可能为包的方法写下 Java 的方法头，并将它们组织为一个 Java 接口，让实现 ADT 的类来使用。程序清单 1-1 中的 Java 接口含有 ADT 包的方法及描述它们行为的详细注释。回想一下，类接口中不含有数据域、构造方法、私有方法或保护方法。

现在，包中的项将是同一个类的对象。例如，我们可以有字符串的包。为能容纳类类型的项，包的方法中使用泛型 (generic data type) T 来表示每个项。必须在接口名的后面写 `<T>`，来说明标识符 T 的含义。一旦客户程序中选择了实际的数据类型，编译程序将在 T 出现的所有地方使用那个数据类型。本章后面的 Java 插曲 1 将讨论如何使用泛型，从而为 ADT 中涉及的数据提供类型的灵活性。

当检查接口时，要留意前一段中提到的处理特殊情形时所做的决策。具体来说，对于 `add`、`remove` 及 `contains` 方法，它们每一个都要返回一个值。因为我们的程序设计语言是 Java，所以要注意，有一个 `remove` 方法返回一个指向项的引用，而不是项本身。

虽然不一定要在实现类之前写接口，但这样做能让你以简洁的方式记下你的规范说明。然后可以将接口中的代码用在具体类的概要设计中。有了接口还能为包提供数据类型，这个类型不依赖于特定的类定义。接下来的两章将开发包类的两种不同实现。针对接口所写的代码，能让我们更易于将包的一种实现替换为另一种。

程序清单 1-1 用于包类的 Java 接口

```

1 /**
2  * An interface that describes the operations of a bag of objects.
3  * @author Frank M. Carrano, Timothy M. Henry
4 */
5 public interface BagInterface<T>
6 {
7     /** Gets the current number of entries in this bag.
8      * @return The integer number of entries currently in the bag. */
9     public int getCurrentSize();
10
11    /** Sees whether this bag is empty.
12     * @return True if the bag is empty, or false if not. */
13    public boolean isEmpty();
14
15    /** Adds a new entry to this bag.
16     * @param newEntry The object to be added as a new entry.
17     * @return True if the addition is successful, or false if not. */
18    public boolean add(T newEntry);
19
20    /** Removes one unspecified entry from this bag, if possible.
21     * @return Either the removed entry, if the removal

```

```

22     was successful, or null. */
23 public T remove();
24
25     /** Removes one occurrence of a given entry from this bag, if possible.
26      @param anEntry The entry to be removed.
27      @return True if the removal was successful, or false if not. */
28 public boolean remove(T anEntry);
29
30     /** Removes all entries from this bag. */
31 public void clear();
32
33     /** Counts the number of times a given entry appears in this bag.
34      @param anEntry The entry to be counted.
35      @return The number of times anEntry appears in the bag. */
36 public int getFrequencyOf(T anEntry);
37
38     /** Tests whether this bag contains a given entry.
39      @param anEntry The entry to find.
40      @return True if the bag contains anEntry, or false if not. */
41 public boolean contains(T anEntry);
42
43     /** Retrieves all entries that are in this bag.
44      @return A newly allocated array of all the entries in the bag.
45      Note: If the bag is empty, the returned array is empty. */
46 public T[] toArray();
47 } // end BagInterface

```

规范说明一个 ADT 并且为它的操作写了 Java 接口后，应该写几个使用 ADT 的 Java 语句。虽然还不能执行这些语句——毕竟我们没写实现 `BagInterface` 的类，但可以用它们来确认或修改方法的设计决策及相关文档。这种方式下，可以检查规范说明的适应性及对它的理解。现在修改 ADT 的设计或文档，好过等到写完实现后再修改。认真做这件事的额外好处是，过后可以使用这些相同的 Java 语句来测试你的实现。

1.14



学习问题 6 给定学习问题 1 中创建的包 `aBag`，写 Java 语句，显示 `aBag` 中所有的字符串。不要改变 `aBag` 的内容。



程序设计技巧：在实现一个类之前写测试程序

写 Java 语句来测试一个类的方法，将有助于你完全理解方法的规范说明。显然，在能正确实现方法之前必须要理解它。如果你还是类的设计者，那么用一用这个类可能有助于你对设计或对文档进行理想的修改。如果在实现类之前做这些修改，会节省时间。因为早晚都要写一个程序来测试你的实现，那么为什么不现在就来写而获益，而非要放到以后再写呢？



注：虽然我们说过，包中的项属于同一个类，但这些项也可能属于因继承关系而相关的类。例如，假定 `Bag` 是实现接口 `BagInterface` 的类。如果我们写下面的语句创建类 `C` 对象的包：

```
BagInterface<C> aBag = new Bag<>();
```

则 `aBag` 中可以包含类 `C` 的对象及 `C` 的任何子类的对象。

下节看看使用包的两个例子。后面，可以用这些示例来测试你的实现。

使用 ADT 包

1.15 设想我们雇一名程序员使用 Java 语言实现 ADT 包，给定到目前为止已有的接口和规范说明。如果假定这些规范说明已经足够清楚，能让程序员完成相关的实现，那么我们可以将 ADT 的这些操作用在程序中，而不需要知道实现的细节。即我们不需要知道程序员如何实现这个包，也能使用这个包。我们只需知道 ADT 包做什么就可以了。本节假定，已经有了一个 Java 类 `Bag`，它实现了程序清单 1-1 给出的 Java 接口 `BagInterface`。下面使用简单的例子来说明我们如何使用 `Bag`。

在程序清单 1-2 的第 13 行，注意到一旦我们选择了包中对象的数据类型——本例中是 `Item`，这个数据类型就包含在接口名后面的尖括号中。还要注意到类名后面是空的尖括号。包中的所有项必须是这个数据类型或是这个数据类型的子类型。编译程序强制我们遵守这条约定。如果是基本数据类型，则可以将对应的包装类的实例放到包中。例如，不是使用基本数据类型 `int` 的实例，而是使用包装类 `Integer` 的实例。

1.16  **示例：在线购物。**当在线购物时，你挑选的商品保存在购物车或购物袋内，直到你准备去结账为止。实现购物网站的程序可以使用类 `Bag` 来维护购物车。毕竟，你挑选的待购物品的次序是不重要的。程序清单 1-2 显示这样一个程序的简单示例。

程序清单 1-2 在线购物中购物袋的维护程序

```

1  /**
2   * A class that maintains a shopping cart for an online store.
3   * @author Frank M. Carrano, Timothy M. Henry
4  */
5  public class OnlineShopper
6  {
7      public static void main(String[] args)
8      {
9          Item[] items = {new Item("Bird feeder", 2050),
10                  new Item("Squirrel guard", 1547),
11                  new Item("Bird bath", 4499),
12                  new Item("Sunflower seeds", 1295)};
13      BagInterface<Item> shoppingCart = new Bag<>();
14      int totalCost = 0;
15
16      // Statements that add selected items to the shopping cart:
17      for (int index = 0; index < items.length; index++)
18      {
19          Item nextItem = items[index]; // Simulate getting item from shopper
20          shoppingCart.add(nextItem);
21          totalCost = totalCost + nextItem.getPrice();
22      } // end for
23
24      // Simulate checkout
25      while (!shoppingCart.isEmpty())
26          System.out.println(shoppingCart.remove());
27
28      System.out.println("Total cost: " + "\t$" + totalCost / 100 + "."
29                          + totalCost % 100);
30  } // end main
31 } // end OnlineShopper

```

输出

| | |
|-----------------|---------|
| Sunflower seeds | \$12.95 |
| Bird bath | \$44.99 |

| | |
|----------------|---------|
| Squirrel guard | \$15.47 |
| Bird feeder | \$20.50 |
| Total cost: | \$93.91 |

为使示例简单，我们创建 `Item` 对象的数组来表示购物者挑选的商品。可在本书的在线资源中找到类 `Item`，它定义了用来描述商品及价格的数据域，还定义了访问这些域的访问方法及 `toString` 方法。

初始时，我们使用 `Bag` 的默认构造方法创建 `Item` 对象的空包。注意到，`shoppingCart` 的数据类型是 `BagInterface<Item>`。这个声明要求 `shoppingCart` 仅能调用声明在 `BagInterface` 中的方法。另外，我们可以用实现 `BagInterface` 的其他类来替换 `Bag`，而不需要修改程序中随后的语句。

要注意将挑选的商品添加到包中的循环，以及结账时一次删除一个的循环。

 **学习问题 7** 在前面的例子中，在结账过程中执行 `while` 循环，直到包空时为止。可以用什么样的 `for` 语句来替换这个 `while` 语句？只根据 `shoppingCart` 的存在与否来写，而不判定数组 `items`。

 **示例：扑满。**你或许有一个扑满、存钱罐，或其他某个容器来存放你剩下的硬币。扑满装硬币，但并不组织它们。而扑满里肯定有重复的硬币。扑满像是一个包，但更简单，因为它仅有 3 个操作：可以将一个硬币添加到扑满中，从扑满中删除一个硬币（摇晃扑满，所以没办法控制哪个硬币掉下来），或看看扑满是否为空。

假定你有表示硬币的类 `Coin`，我们可以创建程序清单 1-3 中给出的类 `PiggyBank`。`PiggyBank` 对象将其硬币保存在包中，即保存在实现了接口 `BagInterface` 的类的实例中。`PiggyBank` 的 `add`、`remove` 和 `isEmpty` 方法分别调用包的方法来得到各自的结果。类 `PiggyBank` 是适配器类的一个示例。关于适配器类详见附录 C。

程序清单 1-3 扑满的类

```

1  /**
2   * A class that implements a piggy bank by using a bag.
3   * @author Frank M. Carrano, Timothy M. Henry
4  */
5  public class PiggyBank
6  {
7      private BagInterface<Coin> coins;
8
9      public PiggyBank()
10     {
11         coins = new Bag<>();
12     } // end default constructor
13
14     public boolean add(Coin aCoin)
15     {
16         return coins.add(aCoin);
17     } // end add
18
19     public Coin remove()
20     {
21         return coins.remove();
22     } // end remove
23 }
```

1.17

```

24     public boolean isEmpty()
25     {
26         return coins.isEmpty();
27     } // end isEmpty
28 } // end PiggyBank

```

1.18 程序清单 1-4 提供了类 PiggyBank 的主要示例。程序将一些硬币添加到扑满中，然后再删除所有的。因为程序没有记录添加到扑满中的硬币，所以没办法控制删除哪个硬币。虽然输出的内容表示从扑满中拿走硬币的次序与它们放到扑满中的次序刚好相反，但这个次序依赖于包的实现。我们在下一章将考虑这些实现。

注意到，除了 main 方法外，程序定义了另外一个方法 addCoin。因为 main 是静态的且调用 addCoin，所以 addCoin 也必须是静态的。方法 addCoin 接收的参数是一个 Coin 对象和一个 PiggyBank 对象。然后方法将硬币添加到扑满中。

程序清单 1-4 类 PiggyBank 的示例

```

1  /**
2   * A class that demonstrates the class PiggyBank.
3   * @author Frank M. Carrano, Timothy M. Henry
4  */
5 public class PiggyBankExample
6 {
7     public static void main(String[] args)
8     {
9         PiggyBank myBank = new PiggyBank();
10
11        addCoin(new Coin(1, 2010), myBank);
12        addCoin(new Coin(5, 2011), myBank);
13        addCoin(new Coin(10, 2000), myBank);
14        addCoin(new Coin(25, 2012), myBank);
15
16        System.out.println("Removing all the coins:");
17        int amountRemoved = 0;
18
19        while (!myBank.isEmpty())
20        {
21            Coin removedCoin = myBank.remove();
22            System.out.println("Removed a " + removedCoin.getCoinName() + ".");
23            amountRemoved = amountRemoved + removedCoin.getValue();
24        } // end while
25        System.out.println("All done. Removed " + amountRemoved + " cents.");
26    } // end main
27
28    private static void addCoin(Coin aCoin, PiggyBank aBank)
29    {
30        if (aBank.add(aCoin))
31            System.out.println("Added a " + aCoin.getCoinName() + ".");
32        else
33            System.out.println("Tried to add a " + aCoin.getCoinName() +
34                           ", but couldn't");
35    } // end addCoin
36 } // end PiggyBankExample

```

输出

```

Added a PENNY.
Added a NICKEL.
Added a DIME.
Added a QUARTER.

```

```
Removing all the coins:  
Removed a QUARTER.  
Removed a DIME.  
Removed a NICKEL.  
Removed a PENNY.  
All done. Removed 41 cents.
```

注：方法可以改变作为参数传给它的对象的状态

传给方法 `addCoin` 的有两个参数：一个硬币和一个扑满。这两个参数都是 `main` 方法中已存在的对象的引用。方法 `addCoin` 将这些引用的备份保存在参数中，你应该记得，它们的行为像是局部变量。虽然 `addCoin` 不能改变引用，因为它们已存在于 `main` 方法中，但它能改变所指对象的状态。具体来说，它通过向扑满中添加硬币从而改变了扑满，即 `PiggyBank` 对象。记住，这个扑满只局限于 `main`，而在 `addCoin` 的外面。

注：在后续章节中一旦实现了包类，你就能实际运行前一个程序清单中给出的程序了。你只需要将类名 `Bag` 替换为后续章节中示例使用的一个类名即可。

注：`javadoc` 标签 `@author`。附录 A 的段 A.8 提到，`javadoc` 标签 `@author` 应该出现在所有类及接口中，用来指明编写这段代码的程序员的名字。尽管在我们为你提供的在线源代码中会出现很多这个标签，但在本书后续的章节中我们并不使用它，为的是节省版面。不过，你应该在你的程序作业中使用它。

学习问题 8 考虑程序清单 1-4 中的程序。在创建类 `PiggyBank` 的实例 `myBank` 后，假定将几个未知硬币加到 `myBank` 中。写代码，从扑满中删除硬币，直到或是删除一分钱硬币，或是扑满为空时为止。

像使用自动贩卖机一样使用 ADT

设想你站在一台自动贩卖机前，如图 1-3 所示，然后从贩卖机中买些东西！

当站在自动贩卖机前时，你会看到它的界面。插入一张信用卡 / 借记卡，并按下实际的按钮或触摸屏按钮就能购物了。下面是对自动贩卖机的观察结果：

- 你仅能执行机器的界面提供给你的规定任务。
- 你必须理解这些任务，即你必须知道买一瓶汽水应该怎么办。
- 你不能访问机器内部，因为锁着的外壳封装了它。
- 即使你不知道内部将发生什么，照样可以使用机器。
- 如果有人用改进版替换了机器内部的机制但没改变界面，你仍能用同样的方式使用机器。

1.19



图 1-3 一台自动贩卖机

与自动贩卖机的用户一样，你就像是本章前面见过的 ADT 包的客户。刚刚说的对自动贩卖机用户的观察，类似于下面对包的客户的观察：

- 客户仅能执行 ADT 包规范说明的操作。这些操作常常声明在一个 Java 接口内。
- 客户必须遵守 ADT 包提供的操作规范。即客户的程序员必须理解如何使用这些操作。
- 客户不使用 ADT 操作就不能访问包内的数据。封装原理将数据表示隐藏在 ADT 的内部。
- 客户可以使用包，即使程序员不知道数据是如何存储的。
- 如果有人改变了包操作的实现，只要界面没有改变，客户仍能用同样的方式使用包。

1.20

在前一节的示例中，每个包都是实现 ADT 包的类的一个实例。即每个包是一个对象，它的行为是 ADT 包的操作。你可以把每个这样的对象看作我们刚描述的自动贩卖机。每个对象封装了包的数据和操作，就像是自动贩卖机封装了它的产品（汽水）和输送系统一样。

有些 ADT 操作有输入，类似于你插入自动贩卖机中的借记卡。有些 ADT 操作有输出，类似于自动贩卖机上提供的罐装汽水、消息及提示灯。

现在设想你是自动贩卖机面板或界面的设计人员。机器能做什么，及在使用机器时人应该做什么？考虑在机器内如何保存及输送罐装汽水，对你是否有帮助？我们强调，你应该忽略这些侧面，而把注意力完全集中于人如何使用机器上，即你要关注界面的设计。忽略无关细节能使你的任务更简单，并提高设计质量。

回忆一下，作为设计原则，抽象要求你关注于什么而不是如何。当你设计一个 ADT，并最终设计一个类时，使用数据抽象将关注焦点集中在你想对数据做什么，而不是担心如何完成这些任务。在本章的开头，当设计 ADT 包时我们练习了数据抽象。当我们挑选包应该含有的方法时，没有考虑如何表示包。相反我们集中考虑每个方法应该做什么。

最后，我们写了一个 Java 接口，其中详细地规范说明了各个方法。然后在仍然不知道它的实现细节的情况下，可以编写一个使用包的客户程序。如果有人为我们写了包的实现，则我们的程序大概能正确执行。如果其他人给我们一个更好的实现版本，则我们不需要修改已经写好的客户程序仍能继续使用。客户的这个特征是抽象的主要优势。

ADT 集合

1.21

集合（set）是一类特殊的包，它不允许有重复项。每当对数据集中的一个项仅需处理一次时，可以使用集合。例如，编译程序必须找到程序中的标识符，并确保每个标识符仅定义了一次。它将遇到的每个标识符添加到一个集合中。如果本次添加不成功，则编译程序就会发现之前创建过的一个标识符。

为了规范说明这个 ADT，我们回过头来看看包的接口。包中的大多数操作与 ADT 集合中的操作一样；但是，我们需要修改 `add` 和 `remove` 的规范说明。而且我们真的不需要 `getFrequencyOf` 操作，因为对于集合，它总是返回 0 或 1。虽然这个结果会告诉我们集合中是否含有给定的项，但我们可以替换使用 `contains` 方法。程序清单 1-5 含有 ADT 集合的接口。不带注释的那几个方法与程序清单 1-1 中为 `BagInterface` 给出的规范说明一样。

程序清单 1-5 集合类的 Java 接口

```

1  /** An interface that describes the operations of a set of objects. */
2  public interface SetInterface<T>
3  {
4      public int getCurrentSize();
5      public boolean isEmpty();
6
7      /** Adds a new entry to this set, avoiding duplicates.
8       * @param newEntry The object to be added as a new entry.
9       * @return True if the addition is successful, or
10          false if the item already is in the set. */
11     public boolean add(T newEntry);
12
13    /** Removes a specific entry from this set, if possible.
14     * @param anEntry The entry to be removed.
15     * @return True if the removal was successful, or false if not. */
16    public boolean remove(T anEntry);
17
18    public T remove();
19    public void clear();
20    public boolean contains(T anEntry);
21    public T[] toArray();
22 } // end SetInterface

```

Java 类库：接口 Set

正如我们在附录 B 的结尾处提到的，Java 类库中包含了类和接口，这是 Java 程序员理所当然会使用的。我们时不时会介绍 Java 类库中与当前的讨论相关的一些成员。**Java 集合框架**（Java Collections Framework）是这个库的一个子库，为我们提供了表示及处理集合的统一方式。我们将要说明的 Java 类库中的许多类和接口，都是这个框架的一部分，不过我们通常不会提到这个事实。

最后，我们介绍标准接口 **Set**，它属于 Java 类库中的包 `java.util`。遵循这个接口规范说明的集合，不含有由相等的 `x` 和 `y` 组成的对象对，即对象相等是指 `x.equals(y)` 为真。1.22

声明在接口 **Set** 中的下列方法头类似于 **SetInterface** 中的方法。我们标出了 **Set** 中方法与 **SetInterface** 中对应方法的不同之处。

```

public boolean add(T newEntry)
public boolean remove(Object anEntry)
public void clear()
public boolean contains(Object anEntry)
public boolean isEmpty()
public int size()
public Object[] toArray()

```

接口 **Set** 和 **SetInterface** 中都声明了若干对方所没有的方法。

本章小结

- 抽象数据类型（ADT）是数据集和数据上操作的规范说明。这个规范说明不指明如何保存数据或如何实现操作，它独立于任何一种程序设计语言。
- 当使用数据抽象来设计一个 ADT 时，要集中在想对数据做什么上，而不用担心如何完成这些任务，即忽略如何表示数据及如何操纵数据的细节。

- 程序设计语言中 ADT 的表示封装了数据和操作。这样处理的结果是，具体的数据表示及方法实现都对客户隐藏。
- 集合（collection）是保存一组其他对象的对象。
- 包是无特定次序的项的有限集合。
- 客户仅能使用 ADT 包中定义的操作来控制或访问包的项。
- 当向包中添加对象时，不能指示项在包中的位置。
- 可以从包中按给定值删除一个对象，或删除一个未指定对象。还可以从包中删除所有的对象。
- 包可以报告它是否含有给定的对象，还可以报告给定对象在包中出现的次数。
- 包可以告诉你它当前含有的对象个数，能提供保存这些对象的数组。
- 集合（set）是一个不含有重复项的包。
- 对要讨论的类，要在实现它们之前使用像 CRC 卡和 UML 表示这样的工具仔细规范说明类中的方法。
- 设计了 ADT 初稿后，通过写一些使用 ADT 的伪代码，确认你理解了操作及它们的设计。
- 应该规范说明遇到特殊情况时方法应该采取的动作。
- 组织 ADT 规范说明的一种方式是写一个 Java 接口。
- 在定义类之前写一个测试它的程序，看看你是否完全理解并满意类中方法的规范说明。

程序设计技巧

- 在实现一个类之前写一个测试程序。写 Java 语句来测试一个类的方法，将有助于你完全理解方法的规范说明。显然，在能正确实现方法之前必须要理解它。如果你还是类的设计者，那么用一用这个类可能有助于你对设计或对文档进行理想的修改。如果在实现类之前做这些修改，会节省时间。因为早晚都要写一个程序来测试你的实现，那么为什么不现在就来写而获益，而非要放到以后再写呢？

练习

1. 给出程序清单 1-3 中类 PiggyBank 的每个方法的规范说明，说明方法的目的，描述方法的参数，写前置条件、后置条件和方法头的伪代码。然后写一个用于这些方法的 Java 接口，其中包括 javadoc 风格的注释。
2. 假定 groceryBag 是一个包，保存着表示不同杂货名字的 10 个字符串。写 Java 语句，统计 groceryBag 中 "soup" 出现的次数并全部删除。不要从包中删除任何其他的字符串。报告包中出现的 "soup" 的个数。groceryBag 中也有可能不含有 "soup"。
3. 给定如练习 2 中所描述的 groceryBag，对 groceryBag 进行操作 groceryBag.toArray()，会有什么结果？
4. 给定如练习 2 中所描述的 groceryBag，写 Java 语句，创建这个包中保存的不同字符串的数组。即如果 "soup" 在 groceryBag 中出现 3 次，则它在数组中仅应该出现一次。数组创建完成后，groceryBag 的内容应该不变。
5. 两个集合的并（union）是将它们的内容合并到一个新集合中。在用于 ADT 包的 BagInterface 接口中添加一个方法 union，它返回由接收调用方法的包和方法参数的包的并得到的一个新包。包含对方法进行充分规范说明的足够的注释。

注意，两个包的并可能含有重复项。例如，如果对象 `x` 在一个包中出现 5 次，在另一个包中出现 2 次，则这两个包的并中 `x` 有 7 次。具体来说，假定 `bag1` 和 `bag2` 都是 `Bag` 对象，这里，`Bag` 实现了 `BagInterface`；`bag1` 中含有 `String` 对象 `a`、`b` 和 `c`；而 `bag2` 中含有 `String` 对象 `b`、`d` 和 `e`。则执行语句

```
BagInterface<String> everything = bag1.union(bag2);
```

后，包 `everything` 中含有字符串 `a`、`b`、`b`、`b`、`c`、`d` 和 `e`。注意，并操作不影响 `bag1` 和 `bag2` 的内容。

6. 两个集合的交（intersection）是由在两个集合中都出现的项组成的新集合。即它含有重叠部分的项。在用于 ADT 包的 `BagInterface` 接口中添加一个方法 `intersection`，它返回由接收调用方法的包和方法参数的包的交得到的一个新包。包含对方法进行充分规范说明的足够的注释。

注意，两个包的交可能含有重复项。例如，如果对象 `x` 在一个包中出现 5 次，在另一个包中出现 2 次，则这两个包的交中 `x` 有 2 次。具体来说，假定 `bag1` 和 `bag2` 都是 `Bag` 对象，这里，`Bag` 实现了 `BagInterface`；`bag1` 含有 `String` 对象 `a`、`b` 和 `c`；而 `bag2` 中含有 `String` 对象 `b`、`b`、`d` 和 `e`。执行语句

```
BagInterface<String> commonItems = bag1.intersection(bag2);
```

后，包 `commonItems` 中仅含有字符串 `b`。如果 `b` 在 `bag1` 中出现 2 次，则 `commonItems` 中将含有 2 个 `b`，因为 `bag2` 也含有 2 个 `b`。注意，`intersection` 操作不影响 `bag1` 和 `bag2` 的内容。

7. 两个集合的差（difference）是在一个集合中删除第二个集合中也出现的项后剩余的项组成的新集合。在用于 ADT 包的 `BagInterface` 接口中添加一个方法 `difference`，它返回由接收调用方法的包和方法参数的包的差得到的一个新包。包含对方法进行充分规范说明的足够的注释。

注意，两个包的差可能含有重复项。例如，如果对象 `x` 在一个包中出现 5 次，在另一个包中出现 2 次，则这两个包的差中 `x` 有 3 次。具体来说，假定 `bag1` 和 `bag2` 都是 `Bag` 对象，这里，`Bag` 实现了 `BagInterface`；`bag1` 含有 `String` 对象 `a`、`b` 和 `c`；而 `bag2` 中含有 `String` 对象 `b`、`b`、`d` 和 `e`。执行语句

```
BagInterface leftOver1 = bag1.difference(bag2);
```

后，包 `leftOver1` 中含有字符串 `a` 和 `c`。执行语句

```
BagInterface leftOver2 = bag2.difference(bag1);
```

后，包 `leftOver2` 中含有字符串 `b`、`d` 和 `e`。注意，`difference` 不影响 `bag1` 和 `bag2` 的内容。

8. 写代码完成下列任务：考虑两个能含有字符串的包。一个包名是 `letters`，含有几个单字符的字符串。另一个包是空包，名为 `vowels`。每次从 `letters` 中删除一个字符串。如果字符串中含有一个元音，则将它放到包 `vowels` 中；否则，丢弃这个串。在检查完 `letters` 中的所有字符串后，报告包 `vowels` 中元音的个数，及包中每个元音出现的次数。

9. 写代码完成下列任务：考虑 3 个能含有字符串的包。一个包名是 `letters`，含有几个单字符的字符串。另一个包名为 `vowels`，含有 5 个字符串，每一个是一个元音。第三个包是空包，名为 `consonants`。每次从 `letters` 中删除一个字符串。检查该字符串是否在包 `vowels` 中出现。如果是，则丢弃这个串。否则将它放到包 `consonants` 中。在检查完 `letters` 中的所有字符串后，报告包 `consonants` 中辅音的个数及包中每个辅音出现的次数。

项目

- 如段 1.21 中所述，一个集合是一个不允许有重复值的特殊包。假定类 `Set<T>` 实现了 `SetInterface<T>`。给定一个空集合，它是 `Set<String>` 的对象，且给定类 `Bag<String>` 的一个对象，

其中含有几个字符串，写客户语句，从给定的包创建一个集合。

2. 假定桌上有一摞书。每本书太大太重，你只能拿走这摞书中最上面的一本。不能从其他书下拿走一本。类似地，你不能在其他书的下面增加一本。增加书时，只能在这摞书的最上面放一本。

如果仅用书名表示一本书，设计一个类，用来记录桌上摞着的书。规范说明每个方法，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于书堆方法的 Java 接口。在代码中包含 `javadoc` 风格的注释。

3. 环 (ring) 是项的集合，它有一个指向当前项的引用。操作——估且称之为 `advance`——将引用移向集合中的下一项。当引用到达最后一项时，下一次 `advance` 操作将引用移回第一项。环还有其他的操作，包括得到当前项、添加一项及删除一项。操作中项添加的位置及删除哪一个项这样的细节由你来决定。

设计一个 ADT 来表示对象的环。规范说明每个方法，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于环方法的 Java 接口。在代码中要包含 `javadoc` 风格的注释。

4. 一个扑克牌盒 (shoe) 中放有若干整副牌。盒中的这些牌可以混洗，然后一次打出一张。也可以计算盒中纸牌的张数。

当一把牌打完，应该将所有的纸牌放回盒中并洗牌。有些纸牌游戏要求，当牌盒变空时，弃牌堆中的牌要放回盒中。然后重洗盒中的牌。本例中，不是所有的牌都在盒中；有些牌在玩家手中拿着。

设计一个 ADT 盒，假定你已有类 `PlayingCard`，这个你也应该规范说明。不需要一副牌的 ADT，因为一副牌就是牌数为 1 的一盒牌。

规范说明每个 ADT 操作，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于盒的方法的 Java 接口。在代码中包含 `javadoc` 风格的注释。

5. 安装空调的一次投标包括公司名、设备描述、设计性能、设备价格及安装费用。

设计表示任意投标的一个 ADT。然后设计另外一个 ADT，表示投标集合。第二个 ADT 应该包含根据价格和性能查找投标的方法。还要注意，一个公司能投多个标，每个标有不同的设备。

规范说明每个 ADT 操作，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于投标方法的 Java 接口。在代码中包含 `javadoc` 风格的注释。

6. 一个矩阵是数值的一个矩形数组。可以将两个矩阵相加或相乘得到第三个矩阵。可以用矩阵乘上一个数量，可以转置矩阵。设计一个表示有这些操作的矩阵的 ADT。

规范说明每个 ADT 操作，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于矩阵的 Java 接口。在代码中包含 `javadoc` 风格的注释。

7. 练习 5、练习 6 和练习 7 要求你规范说明返回两个包的并、交及差的 ADT 包中的方法。独立于包的实现，仅使用 ADT 包操作来定义这些方法。

8. 假定一排汽车在停车场的出口排队。排在最前面的汽车停在收费亭那里。汽车只能在队尾排到当前队列的最后一辆车之后。栅栏及狭窄的空间，使得汽车不能插队。

设计一个 ADT，用来记录排队的汽车。规范说明每个操作，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于排队方法的 Java 接口。在代码中包含 `javadoc` 风格的注释。

9. (游戏) 洞穴系统是一组相互连接的地下隧道。两个或三个隧道相交形成一个洞穴。设计一个用于洞穴和洞穴系统的 ADT。考古学家应该能向洞穴系统中添加一个新发现的洞穴，并用隧道将两个洞穴连通起来。不允许有重复的洞穴——基于 GPS 坐标。考古学家还应该能列出给定洞穴系统中的洞穴。规范说明每个 ADT 操作，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于洞穴方法的 Java 接口及一个用于洞穴系统方法的 Java 接口。在代码中包含 `javadoc` 风格的注释。

10. (财务) 财务账户，如支票账户、信用卡账户和贷款账户都有一些共性。包括购买、支付、退货和额外的利息费用的操作。账户可以提供其当前的交易及余额，并将其数据与月结单进行对账。

- a. 设计一个交易 ADT 和一个财务账户 ADT。财务账户中的数据应该包括顾客名、账户号及账户余额。规范说明每个 ADT 操作，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于交易方法的 Java 接口及一个用于财务账户方法的 Java 接口。在代码中包含 javadoc 风格的注释。
- b. 画一个包括 `Transaction`、`FinancialAccount`、`CreditCardAccount` 及 `CheckingAccount` 在内的 UML 类图。
11. (电子商务) 当在线购物时，你选择商品并放到购物车中。购物车中允许有重复的商品，正如你可以购买多个相同商品一样。如果你改变主意不想买了，你还可以从购物车中删除一件商品。购物车可以展示当前的商品，包括其价格和商品总价。设计商品 ADT 和购物车 ADT。规范说明每个 ADT 操作，说明方法的目的，描述它的参数，写方法头的伪代码。然后写一个用于商品方法的 Java 接口及一个用于购物车方法的 Java 接口。在代码中包含 javadoc 风格的注释。

泛型

先修章节：序言

本书的内容涉及其实例含有数据集的类的设计及创建。任一集合中的数据项都有相同或相关——通过继承——的数据类型。例如，可能有一个字符串集合、`Name` 对象的集合、`Student` 对象的集合，等等。在 Java 中不是为每个这样的集合写一个不同的类，而是允许在类或接口的定义中，用一个占位符替代实际的类类型。基于泛型（generic）的特性，这样做是可行的。通过使用泛型，可以定义一个类，其对象的数据类型由类的客户在以后来确定。这项技术对于学习数据结构很重要，本插曲将告诉你现在要知道的内容。

泛型数据类型

J1.1 泛型能让你在类或接口的定义中写一个占位符，以替代实际的类类型。占位符是泛型数据类型（generic data type），也可以简称为泛型（generic type）或类型参数（type parameter）。当定义一个其实例保存不同的数据集合的类时，不需要给出这些集合中对象的具体数据类型。而是使用泛型数据类型替代实际的数据类型，定义一个泛型类（generic class），由客户来选择集合中对象的数据类型。

正如附录 C 所提到的，对象 `Object` 是所有其他类的最终的祖先。给定指向任意类型对象的一个引用，可以将这个引用赋给 `Object` 类型的变量。虽然能够尝试将 `Object` 用作泛型类，但不应该这样做。而是应该使用泛型数据类型来表示任意的类类型。

假定有对象数组 `A`。如果 `A` 的数据类型声明为 `Object[]`，就可以将对象，比方说字符串放到数组中。但是，没有办法阻止你将几个其他类的对象与字符串一起放到数组中。听上去这或许挺吸引人的，但使用这样的数组可能会有问题。例如，如果从数组中删除一个对象，你不知道它的动态类型是什么。它是字符串还是某个其他类型的对象？不过，可以用方法来获知对象的动态类型，所以这样的数组还是有用武之地的。

相反，由泛型变量指向的项组成的数组或任何其他的组，可能仅含有因继承而相关的类的对象。所以，使用泛型，可以限制集合中项的类型。这个限制很正常，因为它使得这些集合易于使用。

接口中的泛型

J1.2 示例。数学中，有序对（ordered pair）是一对值 a 和 b ，表示为 (a,b) 。我们说， (a,b) 中的值是有序的，因为 (a,b) 不等于 (b,a) ，除非 a 等于 b 。例如，二维空间中的一个点由它的 x 坐标和 y 坐标来表示，即有序对 (x,y) 。

假定有相同类型的对象对。如果每个对本身是一个对象，则我们可以定义一个接口，来描述这样的对的行为，并在它的定义中使用泛型。例如，程序清单 JI1-1 定义了接口 `Pairable`，它规范说明了这些对。`Pairable` 对象含有同一泛型 `T` 的两个对象。

程序清单 JI1-1 接口 Pairable

```

1 public interface Pairable<T>
2 {
3     public T getFirst();
4     public T getSecond();
5     public void changeOrder();
6 } // end Pairable

```

实现这个接口的类的开头可以是下列语句:

J1.3

```
public class OrderedPair<T> implements Pairable<T>
```

这个例子中，在 `implements` 子句中传给接口的数据类型是为本类声明的泛型 `T`。一般来说，可以将实际类的名字传给 `implements` 子句中出现的接口。在 Java 插曲 5 中会看到这样的一个例子。



注：为了在定义接口或类时建立泛型，可以在类头的接口名或类名的后面，写一个用尖括号括起来的标识符——例如 `T`。标识符 `T` 可以是任何的标识符，但通常是单个大写字母。它表示接口或类的定义中的一个引用类型——不是基本类型。

泛型类

程序清单 JI1-2 展示了前一段开始讨论的类 `OrderedPair`。这个类假定，对象对中对象出现的次序是有关系的。符号 `<T>` 接在类头的名字标识符之后。在定义中，`T` 表示两个私有数据域的数据类型、构造方法的两个参数的数据类型、方法 `getFirst` 和 `getSecond` 的返回值类型，方法 `changeOrder` 中局部变量 `temp` 的数据类型。

J1.4

程序清单 JI1-2 类 OrderedPair

```

1 /**
2  * A class of ordered pairs of objects having the same data type.
3 */
4 public class OrderedPair<T> implements Pairable<T>
5 {
6     private T first, second;
7
8     public OrderedPair(T firstItem, T secondItem) // NOTE: no <T> after
9     {                                              // constructor name
10        first = firstItem;
11        second = secondItem;
12    } // end constructor
13
14 /** Returns the first object in this pair. */
15 public T getFirst()
16 {
17     return first;
18 } // end getFirst
19
20 /** Returns the second object in this pair. */
21 public T getSecond()
22 {
23     return second;
24 } // end getSecond
25
26 /** Returns a string representation of this pair. */
27 public String toString()

```

```

28     {
29         return "(" + first + ", " + second + ")";
30     } // end toString
31
32     /** Interchanges the objects in this pair. */
33     public void changeOrder()
34     {
35         T temp = first;
36         first = second;
37         second = temp;
38     } // changeOrder
39 } // end OrderedPair

```



注：在类名字 `<T>` 的定义中，`T` 是泛型类型参数，

- `<T>` 出现在类头的名字标识符之后
- `<T>` 没有出现在定义的构造方法名的后面
- `T`——不是 `<T>`——可以是数据域、方法参数及局部变量的数据类型，它可以是方法的返回值类型

J1.5



示例：创建 `OrderedPair` 对象。例如，要创建 `String` 对象的有序对，可以写如下的语句：

```
OrderedPair<String> fruit = new OrderedPair<>("apple", "banana");
```

现在，`OrderedPair` 定义中作为数据类型出现的 `T`，都将使用 `String` 来替代。



程序设计技巧：在 Java 7 之前，前面这条 Java 语句都需要写两遍数据类型 `String`，如下所示：

```
OrderedPair<String> aPair = new OrderedPair<String>("apple", "banana");
```

现在这种形式也是可以的。

下列语句是如何使用对象 `fruit` 的示例：

```

System.out.println(fruit);
fruit.changeOrder();
System.out.println(fruit);
String firstFruit = fruit.getFirst();
System.out.println(firstFruit + " has length " + firstFruit.length());

```

这些语句得到的输出是：

```
(apple, banana)
(banana, apple)
banana has length 6
```

说明一下，有序对 `fruit` 有 `OrderedPair` 类中的方法 `changeOrder` 和 `getFirst`。另外，`getFirst` 返回的对象是 `String` 对象，它有方法 `length`。

还要说明的是，有些是非法的。不能将不是字符串的对象对赋给 `fruit` 对象：

```
fruit = new OrderedPair<Integer>(1, 2); // ERROR! Incompatible types
```

问题在于不能将 `OrderedPair<Integer>` 转换为 `OrderedPair<String>`。但是可以创建 `Integer` 对象的对，如下所示：

```
OrderedPair<Integer> intPair = new OrderedPair<>(1, 2);
System.out.println(intPair);
intPair.changeOrder();
System.out.println(intPair);
```

输出不出所料：

```
(1, 2)
(2, 1)
```

现在考虑附录 B 的程序清单 B-1 中给出的类 Name。如果变量 namePair 具有类型 OrderedPair<Name>，则能创建 Name 类或是使用继承派生于 Name 的任何类的对象的对。例如，如果类 FormalName 派生于 Name，且增加了一个头衔，如 Mr. (先生) 或 Ms. (女士)，则 namePair 可以含有 Name 和 FormalName 的对象。



注：在泛型类 `name<class-type>` 的客户中，

- 如下形式的表达式

```
new name<class-type>(...)
```

创建了类的对象。从 Java 7 版本起，如果这个表达式赋给一个数据类型是 `name<class-type>` 的变量，则可以省略表达式中的 `class-type`。即可以写如下的语句：

```
name<class-type> var = new name<>(...)
```

- 类对象的数据类型是 `name<class-type>`，而不是 `name`。



学习问题 1 像 String 或 Name 这样的类必须定义哪些方法，才能让 OrderedPair 的方法 `toString` 正常工作？

学习问题 2 考虑程序清单 JI1-2 中所给的类 OrderedPair。假定我们没有使用泛型，而是省略 `<T>`，将私有域、方法参数及局部变量的数据类型声明为 Object 而不是 T。这些修改对类的使用有什么影响？

学习问题 3 你能使用程序清单 JI1-2 中定义的类 OrderedPair，让两个不同及不相关的数据类型的对象配对吗？为什么？

学习问题 4 使用附录 B 的程序清单 B-1 中定义的类 Name，写语句，将两名学生组成实验搭档。

使用数组实现包

先修章节：序言、第 1 章

目标

学习完本章后，应该能够

- 使用定长数组或可动态扩展的数组实现 ADT 包
- 讨论提出的两种实现方案的优缺点

读者已经见过了几个示例，展示了如何在程序中使用 ADT 包。本章提出两种不同的方法——每个都涉及数组——在 Java 中实现一个包。当使用数组来组织数据时，这样的实现称为基于数组的 (array based)。下一章将看到一种完全不同的方法。

我们先使用普通的 Java 数组来表示包中的项。就这种实现方式来说，包可能变为满的，就好像食品杂货袋子也会满一样。然后，我们提出另外一种不受这个问题困扰的实现方式。对于第二种实现，当你用完了数组中的所有空间时，可以将数据移到一个更大的数组中。其效果是，有一个明显扩大了的数组来满足你的需求。所以，我们可以有一个永远也不满的包。

使用定长数组实现 ADT 包

我们的任务是定义在前一章写接口 `BagInterface` 时规范说明的方法。从使用模拟来描述如何用定长数组保存包中的项入手。为此，我们展示 `add` 和 `remove` 方法是如何工作的。随后，给出相应的包的 Java 实现。

模拟

2.1 假定教室（称为教室 A）在固定位置上有 40 把椅子。如果一门课程限制为 30 名学生，则会有 10 把椅子空闲且浪费。如果我们取消选课限制，则即使还有另外 20 名学生想上这个课，也仅能多容纳 10 名学生。

数组就像是这间教室，每把椅子像是一个数组位置。假定我们把教室内的 40 把椅子从 0 开始顺序编号，如图 2-1 所示。尽管在一间典型的教室内椅子按行放置，但我们忽略这个细节，将椅子看成一维数组。

2.2 增加一名新学生。假定老师要求到达的学生坐到已连续编号的椅子上。这样，第一名到达教室的学生坐在 0 号椅子上，第二名学生坐在 1 号椅子上，以此类推。老师提出的坐在已连续编号椅子上的要求是随意给出的，这只是他的习惯。读者会看到，我们将以类似的方式填充包项的数组。

假定教室 A 中的 30 名学生坐在 0 ~ 29 连续编号的椅子上，且有新学生想加入这些学生中。因为教室内有 40 把椅子，所以可占用编号为 30 的椅子。可以简单地把新学生分配到编号 30 的椅子上。当所有 40 把椅子都占满时，教室内不能再容纳更多的学生了。教室满了。

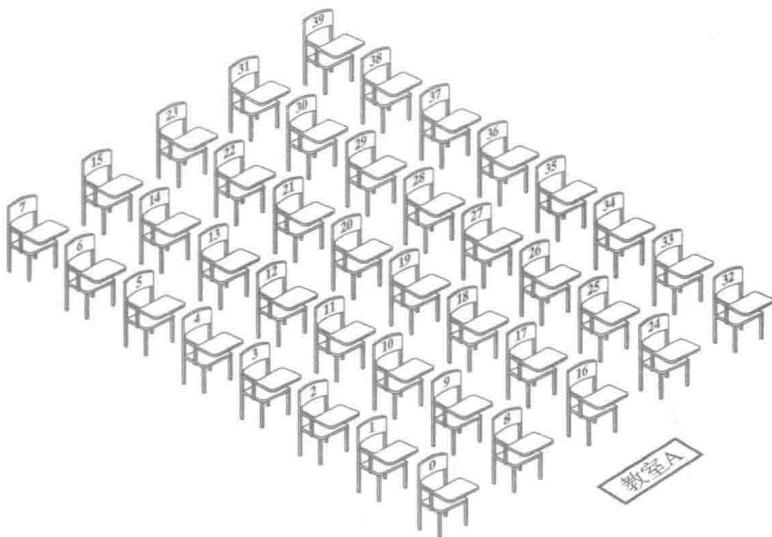


图 2-1 在固定位置放有椅子的教室

删除某名学生。现在假定教室 A 中 5 号椅子上的学生要逃课。5 号椅子在房间内固定的位置上，会空出来。但如果我们仍想让学生坐在连续编号的椅子上，则需要一名学生移到 5 号椅子上。因为学生没有特定的次序，如果坐在最高编号椅子上的学生移到 5 号椅子上，则其他人就不需要再移动了。例如，如果教室内 30 名学生坐在 0 ~ 29 号椅子上，则坐在 29 号椅子上的学生应该移到 5 号椅子上。29 号椅子将空闲出来。

2.3

学习问题 1 刚描述的为让空闲的椅子不再空闲而移动学生的方式有什么优点？



学习问题 2 让空闲下来的椅子空着有什么优点？

学习问题 3 如果学生想逃课，哪个学生逃课不会迫使其他的人换椅子？

一组核心方法

使用 Java 语言基于数组实现的 ADT 包吸收了教室示例中展现出的一些想法。由此得到了类 `ArrayBag`，它实现了第 1 章程序清单 1-1 中见到过的接口 `BagInterface`。接口内的每个公有方法对应于 ADT 包的一个操作。回忆一下，接口为包中的对象定义了泛型 T。我们在 `ArrayBag` 的定义中也用到了这个相同的泛型。

2.4

类 `ArrayBag` 的定义可能相当难懂。类确实有不少的方法。对于这样的类，你不应该定义整个类，然后试图去测试它。而是应该先确定一组核心方法（core method），实现并测试这些方法，然后再继续定义类中的其他部分。将其他方法的定义留待稍后解决，可以集中注意力并简化你的任务。但哪些方法应该属于这组核心方法呢？一般地，这样的方法应该是为达成类的重要目的，且能进行适当的测试。有时称一组核心方法为核心组（core group）。

当处理包这样的集合时，在集合创建之前尚不能测试大多数的方法。所以，将对象添加到集合中就是一个基本操作。如果方法 `add` 没能正确工作，那么测试像 `remove` 这样的方法将是毫无意义的。所以，包的 `add` 方法是我们首先要实现的核心方法组的一部分。

为测试 `add` 是否能正确工作，需要一个能让我们看到包内容的方法。方法 `toArray` 可用于这个目的，所以它是一个核心方法。构造方法也是基本的方法，也在核心组内。同样，核心方法可能调用的其他方法也是核心组的一部分。例如，因为我们不能将项添加到满包

中，所以方法 `add` 通过调用私有方法 `isArrayFull` 来发现一个满数组。

2.5 核心方法。我们已经确定了下列核心方法属于类 `ArrayBag` 的初稿部分：

构造方法

```
public boolean add(T newEntry)
public T[] toArray()
private boolean isArrayFull()
```

有了这些核心方法，我们就能构造一个包、向其中添加对象及查看结果。在这些核心方法能正确工作之前，我们先不实现其他的方法。

 **注：**像 `add` 和 `remove` 这样能改变集合底层结构的方法，可能是与实现方式关系最密切的方法。一般地，这类方法的定义应该先于类中的其他方法。但因为在 `add` 正确之前我们不能测试 `remove`，所以我们将 `remove` 的实现延后到 `add` 完成且进行充分测试之后再进行。

 **程序设计技巧：**当定义一个类时，实现并测试一组核心方法。从向对象集合中添加对象的方法或与实现方式关系最密切的方法入手。

实现核心方法

2.6 数据域。在定义任何核心方法之前，需要考虑类的数据域。因为包要保存一组对象，所以有一个域是这些对象的数组。数组的长度定义了包的容量。可以让客户指定这个容量，我们也可以提供一个默认容量。另外，我们想记录当前包中项的个数。所以可以为我们的类定义如下的数据域：

```
private final T[] bag;
private int numberEntries;
private static final int DEFAULT_CAPACITY = 25;
```

将它们加到前一章图 1-2 中类的 UML 表示中。得到的表示如图 2-2 所示。

 **程序设计技巧：终极数组**

通过声明数组 `bag` 是类 `ArrayBag` 的一个终极数据成员，可知变量 `bag` 中的引用不能再改变。虽然以这种方式声明数组是一个好的做法，但要知道数组中各数组元素 `bag[0]`, `bag[1]`, … 的值还是可以改变的。这样的改变是需要的，但必须保证客户仅能使用 ADT 包的操作来做这些改变，故我们必须阻止客户得到 `bag` 中指向数组的引用。这种情况会让数组的内容容易受恶意毁坏。当在段 2.12 中定义方法 `toArray` 时会进一步讨论这个问题。

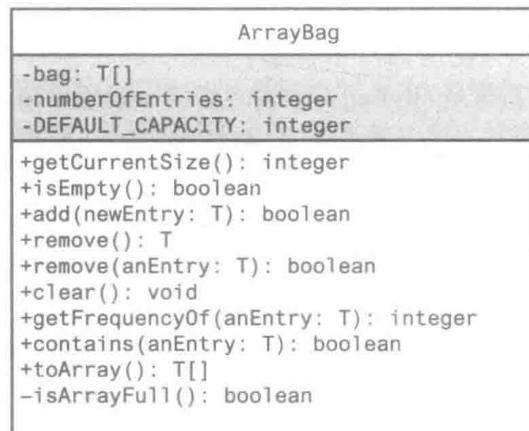


图 2-2 类 `ArrayBag` 的 UML 表示，包括类的数据域

2.7 关于构造方法。这个类的构造方法必须创建数组 `bag`。注意到，前一段中声明数据域

`bag` 时没有创建数组。在构造方法中忘记创建数组是常见错误。为创建数组，构造方法必须指定数组的长度，这是包的容量。因为我们已经创建了一个空包，所以构造方法中还应该将域 `numberOfEntries` 初始化为 0。

决定在数组 `bag` 的声明中使用泛型，影响到我们在构造方法中如何分配这个数组。如下的语句

```
bag = new T[capacity]; // SYNTAX ERROR
```

在语法上是不正确的。当分配数组时不能使用泛型。相反，我们可以分配对象是 `Object` 类型的数组，如下所示：

```
new Object[capacity];
```

但是当试图将这个数组赋给数据域 `bag` 时问题就出现了。语句

```
bag = new Object[capacity]; // SYNTAX ERROR: incompatible types
```

会出现语法错误，因为不能将 `Object[]` 类型的数组赋给 `T[]` 类型的数组，即两个数组的类型不兼容。

转型是必需的，但也会带来自己的问题。语句

```
bag = (T[])new Object[capacity];
```

会出现编译警告

```
ArrayBag.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

如果再次编译这个类，且使用选项 `-Xlint`，则会有更详细的信息，信息的开头如下：

```
ArrayBag.java:24: warning: [unchecked] unchecked cast④  
bag = (T[])new Object[capacity]  
^  
 required: T[]  
 found: Object[]  
where T is a type-variable:  
T extends Object declared in class ArrayBag
```

编译程序想让你来保证将数组中每个项从类型 `Object` 转型为泛型 `T` 都是安全的。因为数组刚刚分配，它含有的是 `null` 项，所以转型是安全的，故我们在有问题的语句之前写如下的注释可让编译程序忽略这个警告：

```
@SuppressWarnings("unchecked")
```

这条给编译程序的命令仅能放在方法定义或变量声明之前。因为赋值语句

```
bag = (T[])new Object[capacity];
```

没有声明 `bag`——`bag` 已经声明过了——故我们将它修改如下：

```
// The cast is safe because the new array contains null entries.  
@SuppressWarnings("unchecked")  
T[] tempBag = (T[])new Object[capacity]; // Unchecked cast  
bag = tempBag;
```

④ 24 是出现问题的语句的行号。

**注：禁止编译警告**

要禁止编译程序给出的未检查转型警告，可以在标记的语句之前写如下的语句：

```
@SuppressWarnings("unchecked")
```

注意，这条命令仅能放在方法定义或变量声明之前。应该包含一条注释，对你禁止编译程序的警告做出解释。

2.8 构造方法。让我们来看看到目前为止所描述的包 `ArrayBag`。在完成了类头和数据域之后，定义了构造方法。程序清单 2-1 中的初始化（第二个）构造方法，根据其参数，即所需要的容量，执行了前一段中展示的步骤。

默认的构造方法使用默认的容量作为参数去调用初始化构造方法。回忆可知，一个构造方法可以使用关键字 `this` 作为方法名，调用同一类中的另一个构造方法。

2.9 在完成构造方法后，可以为公有方法添加注释和头了，从 `BagInterface` 中将这些内容复制过来即可。然后在这些方法头后写空方法。程序清单 2-1 是做了这些步骤之后的结果。下一个任务是实现这 3 个核心方法。

程序清单 2-1 类 ArrayBag 的框架

```

1  /**
2   * A class of bags whose entries are stored in a fixed-size array.
3  */
4  public final class ArrayBag<T> implements BagInterface<T>
5  {
6      private final T[] bag;
7      private int numberEntries;
8      private static final int DEFAULT_CAPACITY = 25;
9
10     /** Creates an empty bag whose initial capacity is 25. */
11     public ArrayBag()
12     {
13         this(DEFAULT_CAPACITY);
14     } // end default constructor
15
16     /** Creates an empty bag having a given initial capacity.
17      @param desiredCapacity  The integer capacity desired. */
18     public ArrayBag(int desiredCapacity)
19     {
20         // The cast is safe because the new array contains null entries.
21         @SuppressWarnings("unchecked")
22         T[] tempBag = (T[])new Object[desiredCapacity]; // Unchecked cast
23         bag = tempBag;
24         numberEntries = 0;
25     } // end constructor
26
27     /** Adds a new entry to this bag.
28      @param newEntry  The object to be added as a new entry.
29      @return True if the addition is successful, or false if not. */
30     public boolean add(T newEntry)
31     {
32         < Body to be defined >
33     } // end add
34
35     /** Retrieves all entries that are in this bag.
36      @return A newly allocated array of all the entries in the bag. */
37     public T[] toArray()
38     {
39         < Body to be defined >

```

```

40     } // end toArray
41
42     // Returns true if the ArrayBag is full, or false if not.
43     private boolean isArrayFull()
44     {
45         < Body to be defined >
46     } // end isArrayFull
47
48     < Similar partial definitions are here for the remaining methods declared in BagInterface. >
49
50     ...
51 } // end ArrayBag

```

! **程序设计技巧：**当定义实现接口的类时，从接口中进行复制来添加类中公有方法的注释和头。这种方式有助于你在实现时检查每个方法的规范说明。另外，之后维护代码的人也容易访问这些规范说明。

设计决策：当数组 bag 中含有部分数据时，包的项应该放在数组的哪些元素中？

当向数组中添加第一个项时，一般将它放在数组的第一个元素中，即下标为 0 的元素。不过这样做也不是必需的，特别是对于实现集合的数组来说。例如，有些集合的实现受益于忽略下标为 0 的数组元素，而将下标 1 作为数组的第一个元素。有时你可能会想先使用数组尾端，然后再使用数组的前面。对于一个包，我们没有理由不按常理做，所以包中的对象将从数组的下标 0 处开始存放。

另一个要考虑的问题是，包的对象是否应该保存在数组连续的元素中？要求 add 方法将对象放到数组 bag 中肯定是合理的，但是为什么我们要关心这个问题呢？这真是一个值得关注的问题吗？关于已规划的实现方案，必须确定下来某些事实或断言，以使每个方法的动作不会对其他方法不利。例如，方法 toArray 必须“知道” add 方法将包的项放在哪里。我们现在的决策会影响到从包中删除一项时将发生什么。方法 remove 需要保证数组项保存在连续的元素中吗？它必须这样做，因为至少到现在，我们仍强调包项要保存在连续的数组元素中。

方法 add。如果包满了，则不能添加任何东西。这种情形下，方法 add 应该返回假。否则，仅需在数组 bag 最后项的后面添加 newEntry，语句如下：

```
bag[numberOfEntries] = newEntry;
```

如果向空包中添加，则 numberOfEntries 将是 0，应该给 bag[0] 赋值。如果包中含有一个项，则添加的项应该赋给 bag[1]，以此类推。每次向包中添加项后，要增大计数器 numberOfEntries 的值。这些步骤如图 2-3 所示，且由下面的 add 方法的定义来完成。

```

/** Adds a new entry to this bag.
 * @param newEntry The object to be added as a new entry.
 * @return True if the addition is successful, or false if not. */
public boolean add(T newEntry)
{
    boolean result = true;
    if (isArrayFull())
    {
        result = false;
    }
    else

```

```
{ // Assertion: result is true here
    bag[numberOfEntries] = newEntry;
    numberOfEntries++;
} // end if
return result;
} // end add
```



图 2-3 向表示包的数组中添加项，包的容量是 6，直到它满时为止

注意，我们调用了方法 `isArrayFull`，就好像它已经定义过了一样。之前我们没有把 `isArrayFull` 作为核心方法，现在使用它表明了它应该在核心组内。



注：包中的项没有特定的次序。所以，方法 `add` 可以将新项放到数组 `bag` 中最方便的元素位置。前面那个 `add` 的定义中，元素紧接在已用的最后元素之后。



注：通常，讨论中提到数组时就好像它真的含有对象一样。实际上，Java 数组含有指向对象的引用，如图 2-3 中的数组一样。

2.11

方法 isArrayFull。当包中含有的对象数与数组 bag 能容纳的量相等时包就满了。

`numberOfEntries` 等于数组容量时即发生了这种情形。所以，`isArrayFull` 有下列简单的定义：

```
// Returns true if the bag is full, or false if not.
private boolean isArrayFull()
{
    return numberOfEntries >= bag.length;
} // end isArrayFull
```

方法 `toArray`。^{2.12} 初始核心组内的最后一个方法 `toArray` 是获取包中的项，并将它们返回到客户新分配的数组内。这个新数组的长度可以与包中项的个数——`numberOfEntries` 的值——相等，而不是与数组 `bag` 的长度相等。但是，在分配数组时遇到了定义构造方法时遇到过的同样问题，所以采用与构造方法相同的处理步骤。

在 `toArray` 创建新数组后，使用简单的循环可以将数组 `bag` 中的引用复制到这个新数组中，然后返回这个数组。所以 `toArray` 的定义如下所示。

```
/** Retrieves all entries that are in this bag.
 * @return A newly allocated array of all the entries in the bag. */
public T[] toArray()
{
    // The cast is safe because the new array contains null entries.
    @SuppressWarnings("unchecked")
    T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast
    for (int index = 0; index < numberOfEntries; index++)
    {
        result[index] = bag[index];
    } // end for
    return result;
} // end toArray
```



设计决策：方法 `toArray` 应该返回数组 `bag` 而不是拷贝吗？

假定我们如下定义 `toArray`：

```
public T[] toArray()
{
    return bag;
} // end toArray
```

这个简单定义肯定能将包元素所在的数组返回给客户。例如，如果 `myBag` 是字符串的包，则语句

```
String[] bagArray = myBag.toArray();
```

能得到指向数组的引用，数组中含有 `myBag` 中的项。客户可以使用变量 `bagArray` 来显示 `myBag` 的内容。

但是，引用 `bagArray` 指向的是数组 `bag` 自身，即 `bagArray` 是对象 `myBag` 内私有实例变量 `bag` 的别名，所以它能让客户直接访问这个私有数据。故客户不用调用类的公有方法就能修改包的内容。例如，如果 `myBag` 是图 2-3 中所示的满包，语句

```
bagArray[1] = null;
```

将把项 `Tia` 改为 `null`。如果本意是想从包中删除 `Tia`，虽然这个方法听上去很好，但这样做可能会破坏包的完整性。具体来说，数组 `bag` 中的项可能不再是连续的，且包中的项数也会出错。



安全说明：方法不应该返回指向类中私有数据域的引用，而是应该返回指向数据域拷贝的引用。



注：使用泛型，可以限制集合中项的数据类型，这是因为：

- 数据类型声明为 `Object` 的变量可以是指向任何类类型对象的引用，但是有泛型数据类型的变量仅能指向指定类类型的对象。
- 由 `Object` 类型的变量指向的项的集合，可以含有各种无关类的对象，但由泛型变量指向的项的集合，只能含有由继承而相关的类的对象。



学习问题 4 在段 2.12 给出的 `toArray` 方法中，一般来讲，`numberOfEntries` 的值等于 `bag.length` 吗？

学习问题 5 假定 `toArray` 方法让新数组 `result` 与数组 `bag` 等长。客户如何得到返回的数组中的项数？

学习问题 6 假定 `toArray` 方法返回数组 `bag` 而不是返回像 `result` 这样的新数组。如果 `myBag` 是含 5 个项的包，则下列语句对数组 `bag` 和域 `numberOfEntries` 的影响是什么？

```
Object[] bagArray = myBag.toArray();
bagArray[0] = null;
```

学习问题 7 如果你调用方法 `Arrays.copyOf`，则方法 `toArray` 的方法体中可以含有一个 `return` 语句。修改方法 `toArray`。

让实现安全

2.13

鉴于当今黑客及对重要软件系统未经授权入侵的现实情况，程序员必须在代码中添加安全措施，以使程序对使用者是安全的。虽然 Java 为你管理内存、检查数组下标的合法性，且是类型安全的，但一个错误会使你的代码易受攻击。实现 ADT 时应该时刻铭记安全性，尽管在已有的代码中增加安全机制可能是困难的。



注：你可以在程序中检查可能出现的错误来练习编写故障安全程序设计 (fail-safe programming)。安全程序设计 (safe and secure programming) 通过验证输入给方法的数据和参数的合法性，消除方法的副作用，对客户和使用者的行为不做任何假设，从而扩展了有安全机制程序的概念。



安全说明：保护 ADT 实现的完整性

当实现一个 ADT 时，必须问自己的两个问题是：

- 如果构造方法没有完全执行可能会发生什么？例如，构造方法可能在完成初始化之前抛出一个异常或错误。但是入侵者可能会捕获异常或错误，并试着使用部分初始化的对象。
- 如果客户试图创建一个其容量超出给定范围的包时可能会发生什么？
- 如果这两个行为可能导致问题，那么我们必须阻止或是解决问题，正如接下来你将看到的。

2.14

对于类 `ArrayBag`，我们想防范前面安全说明中所描述的两种情形。现在开始细化

ArrayBag 的不完整的实现，在类中增加下列两个数据域以使代码更安全：

```
private boolean integrityOK;
private static final int MAX_CAPACITY = 10000;
```

这两个修改都涉及构造方法。因为默认构造方法调用带参数的构造方法，所以仅修改后者就足够了。为确保客户不能创建太大的包，构造方法应该相对于 MAX_CAPACITY 值检查客户所需的包的容量。如果需要的容量太大，则构造方法可以抛出一个异常。

如果所需的容量处在允许范围内，则 ArrayBag 的构造方法为什么还不能正确完成呢？因为内存不足可能导致分配数组失败。这样一个事件会导致错误 OutOfMemoryError。一般地，客户将这个错误看作致命事件。黑客可能捕获这个错误，就像你捕获异常一样，并试图使用部分初始化的对象。为防止这种情况发生，类的每个重要方法在执行其操作之前都可以检查域 integrityOK 的状态。这种方式下，畸形对象就不会再有动作。对于正确初始化的对象，构造方法将把域 integrityOK 置为真。

下面是修改后的构造方法。

```
public ArrayBag(int desiredCapacity)
{
    integrityOK = false;
    if (desiredCapacity <= MAX_CAPACITY)
    {
        // The cast is safe because the new array contains null entries
        @SuppressWarnings("unchecked")
        T[] tempBag = (T[])new Object[desiredCapacity]; // Unchecked cast
        bag = tempBag;
        numberOfEntries = 0;
        integrityOK = true; // Last action
    }
    else
        throw new IllegalStateException("Attempt to create a bag whose " +
                                         "capacity exceeds allowed maximum.");
} // end constructor
```

注意，构造方法在成功完成其他任务后，最后一个动作是将 integrityOK 赋值为真。还注意到，IllegalStateException 是标准运行时异常。Java 插曲 2 将解释如何抛出一个异常。

下面来看看如何使用 integrityOK。

在数组 bag 已成功分配的基础上，ArrayBag 中的任何公有方法在继续执行之前，都应该确保数据域 integrityOK 的值为真。如果 integrityOK 为假，则这样的方法可以抛出一个异常。例如，可以修改方法 add，如下所示。

```
public boolean add(T newEntry)
{
    if (integrityOK)
    {
        boolean result = true;
        if (isArrayFull())
        {
            result = false;
        }
        else
        { // Assertion: result is true here
            bag[numberOfEntries] = newEntry;
            numberOfEntries++;
        } // end if
    }
    return result;
}
```

```

    }
    else
        throw new SecurityException("ArrayBag object is corrupt.");
} // end add

```

 注：异常 `SecurityException` 和 `IllegalStateException` 都是包 `java.lang` 中的标准运行时异常。因此，不需要 `import` 语句。

因为我们在多个方法中都要检查 `integrityOK`，所以为了避免代码重复，可以定义下列私有方法。

```

// Throws an exception if this object is not initialized.
private void checkIntegrity()
{
    if (!integrityOK)
        throw new SecurityException("ArrayBag object is corrupt.");
} // end checkIntegrity

```

则方法 `add` 修改如下：

```

public boolean add(T newEntry)
{
    checkIntegrity();
    boolean result = true;
    if (isArrayFull())
    {
        result = false;
    }
    else
    { // Assertion: result is true here
        bag[numberOfEntries] = newEntry;
        numberofEntries++;
    } // end if
    return result;
} // end add

```

应该以相同的方式修改核心方法 `toArray`，因为它调用了 `ArrayBag` 的数据域 `bag`。

 安全说明：你所熟知的编写 Java 代码时的某些常用准则，实际上提升了代码的安全性。这些准则有：

- 将类的大多数数据域声明为私有的，如果不是全部。任何公有数据域都应该是静态和终极的，且有常量值。
- 避免那些掩盖代码安全性的所谓聪明的逻辑。
- 避免重复代码。相反，将这样的代码封装为一个可供其他方法调用的私有方法。
- 当构造方法调用一个方法时，确保这个方法不能被重写。

 安全说明：终极类。注意到，我们将 `ArrayBag` 声明为一个终极类。因此，从 `ArrayBag` 不能派生出其他的类，即 `ArrayBag` 不能是另一个类的超类或基类。终极类比非终极类更安全，因为程序员不能使用继承来改变它的行为。稍后我们将细化这个做法，定义终极方法而不是整个终极类。

测试核心方法

的其他方法怎么办呢？因为 `ArrayBag`——程序清单 2-1 中给出的——实现了 `BagInterface`，所以 Java 语法检查程序将查看这个接口中声明的每个方法的定义。我们要不要等到完成它们的定义后才开始测试？绝对不要！在你写方法的同时就进行测试，会让你尽早发现逻辑错误。不过，不是写完 `BagInterface` 中每个方法的完整实现，而是对可暂时忽略的方法给出它们的不完整定义。

一个不完整定义的方法称为存根（stub）。存根仅仅是用来应付语法检查器的。例如，对于返回一个值的每个方法，添加一个 `return` 语句让其返回一个哑值，就可以避免语法错误。返回布尔值的方法可以返回假。返回对象的方法可以返回 `null`。另一方面，`void` 方法可以简单到只有一个空方法体。

例如，方法 `remove` 最终返回被删除的项，所以它的存根必须含有一个 `return` 语句，如下所示。

```
public T remove()
{
    return null; // STUB
} // end remove
```

void 方法 `clear` 的存根应该是

```
public void clear()
{
    // STUB
} // end clear
```

注意，如果你想在测试程序中调用存根，则存根应该显示一条信息来报告它被调用过。

! 程序设计技巧：不要等到完全实现 ADT 后才测试它。写了存根（这是所需方法的不完整定义）后，就可以尽早开始测试。

测试程序。程序清单 2-2 所示的程序专门用来测试这一阶段所开发的类 `ArrayBag`^Θ 的核心方法 `add` 和 `toArray`。初始时，`main` 方法使用默认构造方法创建一个空包。因为这个包的容量是 25，所以如果你添加少于 25 个项，数组不应该满。故每次添加后，`add` 方法都应该返回真。实际上，程序描述性的输出信息指明被测的方法是正确的。

随后在 `main` 方法中，我们考虑容量为 7 的包，然后向它添加 7 个字符串。这一次，如果试图进行第 8 次添加，`add` 方法应该返回假。同样，程序描述性的输出信息表明方法是正确的。

程序清单 2-2 测试 `ArrayBag` 类的核心方法的程序

```
1  /**
2   * A test of the constructors and the methods add and toArray,
3   * as defined in the first draft of the class ArrayBag.
4  */
5  public class ArrayBagDemo1
6  {
7      public static void main(String[] args)
8      {
9          // Adding to an initially empty bag with sufficient capacity
10         System.out.println("Testing an initially empty bag with " +
11             " sufficient capacity:");
12     }
13 }
```

^Θ 注意，类 `ArrayBag` 的这个版本已在本书的在线网站中提供，名为 `ArrayBag1`。

```

12     BagInterface<String> aBag = new ArrayBag<>();
13     String[] contentsOfBag1 = {"A", "A", "B", "A", "C", "A"};
14     testAdd(aBag, contentsOfBag1);
15
16     // Filling an initially empty bag to capacity
17     System.out.println("\nTesting an initially empty bag that " +
18         " will be filled to capacity:");
19     aBag = new ArrayBag<>(7);
20     String[] contentsOfBag2 = {"A", "B", "A", "C", "B", "C", "D",
21         "another string"};
22     testAdd(aBag, contentsOfBag2);
23 } // end main
24
25 // Tests the method add.
26 private static void testAdd(BagInterface<String> aBag,
27                             String[] content)
28 {
29     System.out.print("Adding the following strings to the bag:");
30     for (int index = 0; index < content.length; index++)
31     {
32         if (aBag.add(content[index]))
33             System.out.print(content[index] + " ");
34         else
35             System.out.print("\nUnable to add " + content[index] +
36                 " to the bag.");
37     } // end for
38     System.out.println();
39
40     displayBag(aBag);
41 } // end testAdd
42
43 // Tests the method toArray while displaying the bag.
44 private static void displayBag(BagInterface<String> aBag)
45 {
46     System.out.println("The bag contains the following string(s):");
47     Object[] bagArray = aBag.toArray();
48     for (int index = 0; index < bagArray.length; index++)
49     {
50         System.out.print(bagArray[index] + " ");
51     } // end for
52
53     System.out.println();
54 } // end displayBag
55 } // end ArrayBagDemo1

```

输出

```

Testing an initially empty bag with sufficient capacity:
Adding the following strings to the bag: A A B A C A
The bag contains the following string(s):
A A B A C A

```

```

Testing an initially empty bag that will be filled to capacity:
Adding the following strings to the bag: A B A C B C D
Unable to add another string to the bag.
The bag contains the following string(s):
A B A C B C D

```



程序设计技巧：方法的全面测试还应该包括实参取其对应参数合理范围内外的值的情况。

注意到，除了 `main` 方法外，`ArrayBagDemo1` 还有两个其他的方法。因为 `main` 是静态的，且调用这两个方法，所以它们也必须是静态的。方法 `testAdd` 的参数接收一个包和一

个字符串数组。该方法使用循环将数组中的每个字符串添加到包中。它还测试了 `add` 方法的返回值。最后，方法 `displayBag` 的参数是一个包，并使用包的方法 `toArray` 来访问它的内容。一旦我们有一个包项的数组，这个简单的循环就可以显示它们。

 学习问题 8 在 `ArrayBagDemo1` 的 `main` 方法中执行下列语句的结果是什么？

```
ArrayBag<String> aBag = new ArrayBag<>();
displayBag(aBag);
```

实现更多的方法

现在可以向包中添加对象了，着手来实现其余的方法，可以从最简单的方法入手。直到我们明白如何查找一个包时再来定义 `remove` 方法。

方法 `isEmpty` 和 `getCurrentSize`。方法 `isEmpty` 和 `getCurrentSize` 的定义很简单，正如你所见的：

```
public boolean isEmpty()
{
    return numberOfEntries == 0;
} // end isEmpty
public int getCurrentSize()
{
    return numberOfEntries;
} // end getCurrentSize
```

 安全说明：方法应该何时调用 `checkIntegrity`？

方法 `isEmpty` 和 `getCurrentSize` 没有调用 `checkIntegrity`。虽然它们能调用，但我们不想因不必要的安全检查而使客户的性能降低。这两个方法都涉及了数据域 `numberOfEntries`。即使构造方法没有完成它的初始化，因此还没有将这个域设置为 0，Java 也会使用默认值将它初始化为 0。所以，任何已进行部分初始化的包都是空的。对于 `ArrayBag`，访问数组 `bag` 的方法都应该确保它已存在。

 注：有些方法的定义非常简单，几乎和类的早期版本中用来定义它们的存根是一样的。包方法 `isEmpty` 和 `getCurrentSize` 就是这样的情况。虽然这两个方法不在第一批核心方法中，不过它们本来是可以在的。即我们可以更早地定义它们，而不是为它们写存根。

方法 `getFrequencyOf`。^{2.20} 为计算给定对象在包中出现的次数，我们统计对象在数组 `bag` 中出现的次数。使用 `for` 循环，循环处理从 0 到 `numberOfEntries-1` 的下标，将给定对象与数组中的每个对象进行比较。每次发现相等时，计数器加 1。循环结束时，只要返回计数器的值就可以了。注意，比较对象时必须使用方法 `equals`，而不是使用相等操作符 `==`。故必须写语句

```
anEntry.equals(bag[index]) // Compares values
```

而不是写语句

```
anEntry == bag[index] // WRONG! Compares locations (addresses)
```

我们假定对象所属的类中定义了自己的 `equals` 方法。

这个方法定义如下。

```
public int getFrequencyOf(T anEntry)
{
    checkIntegrity();
    int counter = 0;
    for (int index = 0; index < numberOfEntries; index++)
    {
        if (anEntry.equals(bag[index]))
        {
            counter++;
        } // end if
    } // end for
    return counter;
} // end getFrequencyOf
```

2.21 方法 contains。要查看包中是否含有给定的对象，可以再次查找数组 bag。这里需要的循环类似于方法 getFrequencyOf 中使用的，但是一旦发现要找项的第一次出现，循环就应该立刻停止。描述这个逻辑的伪代码如下：

```
while (没找到anEntry, 且还有要检查的数组元素)
{
    if (anEntry 等于下一个数组项)
        在数组中找到anEntry
}
```

这个循环的终止条件有两个：已经在数组中找到 anEntry，或是已经查找了整个数组但没成功。

然后来定义方法 contains。

```
public boolean contains(T anEntry)
{
    checkIntegrity();
    boolean found = false;
    int index = 0;
    while (!found && (index < numberOfEntries))
    {
        if (anEntry.equals(bag[index]))
        {
            found = true;
        }
        else
            index++;
    } // end while
    return found;
} // end contains
```



注：两种循环

为计算数组中项出现的次数，方法 getFrequencyOf 使用一个循环来遍访数组的所有项。事实上，循环体执行了 numberOfEntries 次。相反，为表示一个给定项是否出现在一个数组中，方法 contains 中的循环，一经找到要找的项就立即结束。这个循环的循环体执行的次数为 1 ~ numberOfEntries。你应该能轻松地写出执行确定次数或可变次数的循环。



学习问题 9 方法 contains 可以调用 getFrequencyOf 而不是执行一个循环。即你可以像下面这样定义方法：

```
public boolean contains(T anEntry)
{
    return getFrequencyOf(anEntry) > 0;
} // end contains
```

这个定义与前一段中给出的定义相比，优缺点各是什么？

测试附加的这些方法。在为类 `ArrayBag` 定义了附加方法的同时，应该测试它们。本书的在线网站提供的程序 `ArrayBagDemo2`，仅关注于这些附加的方法。不过，你应该逐步地形成一个测试程序，为的是它能测试到目前为止你已经定义的所有方法。类 `ArrayBag` 到目前为止的版本，可以在在线网站名为 `ArrayBag2` 的源代码中找到。

2.22

删除项的方法

我们推迟到现在才来定义从包中删除项的 3 个方法，因为三者中的一个有些困难，它涉及的查找机制很像是我们在 `contains` 中所用到的。我们从更易定义的另外两个方法入手。

方法 clear。 方法 `clear` 从包中删除所有的项，一次删除一个。下面这个 `clear` 的定义调用方法 `remove`，直到包为空时结束。

2.23

```
/** Removes all entries from this bag. */
public void clear()
{
    while (!isEmpty())
        remove();
} // end clear
```

每次循环中要删除哪个项是不重要的。所以，我们调用的是删除一个未指定项的 `remove` 方法。另外，不保存方法返回的这个项。

因为 `remove` 方法要访问数组 `bag`，它应该调用 `checkIntegrity`，以确保包是存在的。所以 `clear` 不需要显式地调用它。

 **注：**我们可以根据尚未定义的方法 `remove` 来定义方法 `clear`。但是，在定义 `remove` 之前，我们不能完全测试 `clear`。

 **学习问题 10** 修改方法 `clear` 的定义，让它不调用 `isEmpty`。

提示：while 语句应该有一个空循环体。

学习问题 11 用下列语句

```
numberOfEntries = 0;
```

替换段 2.23 所示的 `clear` 中的循环，有什么缺点？

删除未指定的项。只要包不空，不带参数的 `remove` 方法从包中删除一个未指定的项。回忆程序清单 1-1 所示的接口中给出的方法的规范说明，该方法返回被它删除的项：

2.24

```
/** Removes one unspecified entry from this bag, if possible.
 * @return Either the removed entry, if the removal was successful,
 *         or null otherwise. */
public T remove()
```

如果方法执行前包是空的，则返回 `null`。

从包中删除一个项，涉及从数组中删除它。虽然我们能访问数组 `bag` 中的任何项，但最后一项是最容易删除的。为此，要

- 访问最后一项，以便它能被返回
- 将项的数组元素设置为 null
- `numberOfEntries` 减 1

`numberOfEntries` 减 1，就会忽视最后一项，意味着它已被高效地删除了，即使我们没有将它在数组中的位置设置为 null。但是不要跳过这一步。

将前面的步骤转为 Java 程序，得到如下的方法：

```
public T remove()
{
    checkIntegrity();
    T result = null;
    if (numberOfEntries > 0)
    {
        result = bag[numberOfEntries - 1];
        bag[numberOfEntries - 1] = null;
        numberOfEntries--;
    } // end if
    return result;
} // end remove
```



安全说明：将数组元素 `bag[numberOfEntries - 1]` 设置为 null，将被删除对象标记为可进行垃圾回收，并防止恶意代码来访问它。



安全说明：在正确计数后更新计数器。在前面的代码中，删除数组最后一项后才将 `numberOfEntries` 减 1，哪怕表达式 `numberOfEntries - 1` 共计算了 3 次。虽然下面的改进可以避免这个重复，但时间上微不足道的节省，不值得要冒太早减小计数器所带来的不安全风险：

```
numberOfEntries--;
result = bag[numberOfEntries];
bag[numberOfEntries] = null;
```

不可否认，在这种情形下，数组和计数器不同步的情况还是有可能的。不管怎样，如果逻辑更复杂，则数组处理过程中可能会发生异常。这个中断将会导致已更新的计数器不准确。



学习问题 12 为什么方法 `remove` 将从数组 `bag` 中删除的项替换为 null？

学习问题 13 前一个 `remove` 方法删除数组 `bag` 中的最后一项。删除另外的项为什么会更难完成？

2.25

删除指定的项。从包中删除项的第 3 个方法将涉及删除指定的项——称为 `anEntry`。如果项在包中出现多次，则仅删除它的一次出现。没有指定要删除哪次出现。我们只删除查找 `anEntry` 时遇到的首次出现。正如我们在第 1 章段 1.9 中所讨论的，方法将根据在包中是否找到这个项而返回真或假。

假定包不空，则查找数组 `bag` 的过程很像是段 2.21 中 `contains` 方法所做的。如果 `anEntry` 等于 `bag[index]`，则记下 `index` 的值。图 2-4 展示成功查找后的数组。

现在需要删除 `bag[index]` 中的项。如果只写

```
bag[index] = null;
```

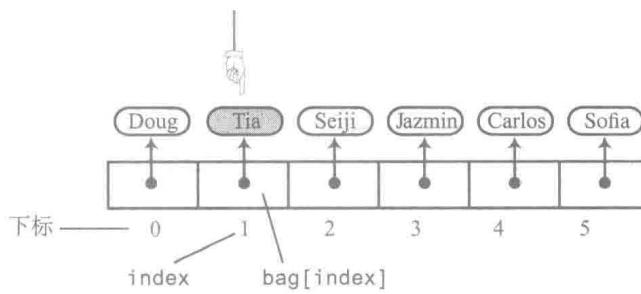
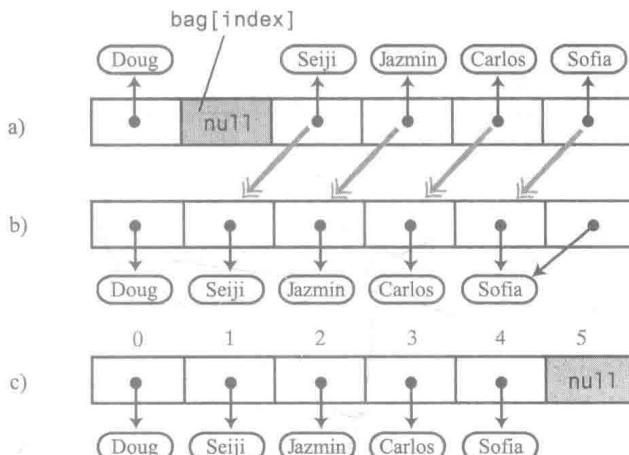


图 2-4 成功查找字符串 "Tia" 后的数组 bag

则只删除了 `bag[index]` 中指向项的引用，但数组中会留下空隙。即包的内容不再占据连续的数组位置，如图 2-5a 所示。我们可以移动随后的项来去掉这个空隙，如图 2-5b 所示，并将指向最后一项的重复引用替换为 `null`，如图 2-5c 所示。但不一定非要用这个费时的方法。

图 2-5 a) 将 `bag[index]` 中的项置为 `null` 后数组 bag 中的空隙；b) 移动随后的项去掉空隙后的数组；c) 将指向最后一项的重复引用替换为 `null` 后

记住，我们不需要维护包中项的具体次序。所以删除一项后，不是移动数组项，而是用数组中最后一项替换被删除的项，如下所示。找到 `bag[index]` 中的 `anEntry` 后，如图 2-6a 所示，将 `bag[numberOfEntries - 1]` 中的项拷贝到 `bag[index]` 中（图 2-6b）。然后将 `bag[numberOfEntries - 1]` 中的项替换为 `null`，如图 2-6c 所示，最后 `numberOfEntries` 减 1。

删除指定项的伪代码。现在将前面的讨论用伪代码写出来，对指定的项 `anEntry`，从含有它的包中删除：

```
在数组bag中找到anEntry; 假定它出现在bag[index]处
bag[index] = bag[numberOfEntries - 1]
bag[numberOfEntries - 1] = null
计数器numberOfEntries减1
return true
```

这段伪代码假定包中含有 `anEntry`。

在伪代码中添加一些细节，以适应 `anEntry` 不在包中的情形，伪代码如下：

```

在数组bag中查找anEntry
if(anEntry在包中且出现在bag[index]处)
{
    bag[index] = bag[numberOfEntries - 1]
    bag[numberOfEntries - 1] = null
    计数器numberOfEntries减1
    return true
}
else
    return false

```

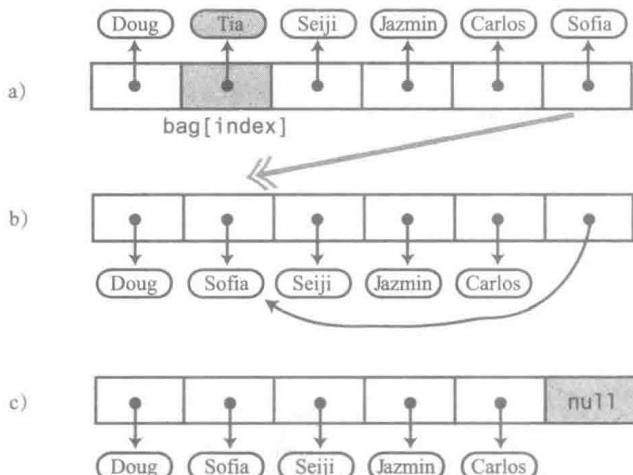


图 2-6 避免删除一项时数组中的空隙

2.27

避免重复工作。很容易将这段伪代码翻译为 Java 方法 `remove`。但是，如果我们这样做了，就会发现新方法和在段 2.24 中写过的 `remove` 方法有很多类似的地方。实际上，如果 `anEntry` 出现在 `bag[numberOfEntries - 1]` 处，则这两个 `remove` 方法将有完全相同的效果。为避免这样的重复劳动，两个 `remove` 方法可以调用一个私有方法来完成删除操作。可以说明如下的一个方法：

```

// Removes and returns the entry at a given array index.
// If no such entry exists, returns null.
private T removeEntry(int givenIndex)

```

因为这是一个私有方法，类内的其他方法可以给它传一个下标作为参数，故仍能让这个下标——实现细节——对类的客户隐藏。

在实现这个私有方法之前，让我们看看是否可以用它来修改段 2.24 中的 `remove` 方法。因为方法删除并返回数组 `bag` 的最后一项，即 `bag[numberOfEntries - 1]`，故它的定义中可以调用 `removeEntry(numberOfEntries - 1)`。继续我们的工作，就如同 `removeEntry` 已经定义且测试过了，可以如下定义 `remove`：

```

/** Removes one unspecified entry from this bag, if possible.
 * @return Either the removed entry, if the removal was successful,
 *         or null otherwise */
public T remove()
{
    checkIntegrity();
    T result = removeEntry(numberOfEntries - 1);
    return result;
} // end remove

```

这个定义看上去不错。我们来实现第二个 `remove` 方法。

第二个 `remove` 方法。第一个 `remove` 方法不查找要删除的项，因为它删除数组中的最后一项。但第二个 `remove` 方法必须执行查找操作。现在先不考虑在数组中查找一个项的细节，我们将这个任务委派给另一个私有方法来完成，它的规范说明如下所示。

```
// Locates a given entry within the array bag.  
// Returns the index of the entry, if located, or -1 otherwise.  
private int getIndexOf(T anEntry)
```

假定这个私有方法已经定义并且测试过了，我们在第二个 `remove` 方法中调用它，如下所示。

```
/** Removes one occurrence of a given entry from this bag.  
 * @param anEntry The entry to be removed.  
 * @return True if the removal was successful, or false if not. */  
public boolean remove(T anEntry)  
{  
    checkIntegrity();  
    int index = getIndexOf(anEntry);  
    T result = removeEntry(index);  
    return anEntry.equals(result);  
} // end remove
```

注意到，`removeEntry` 返回它删除的项，或是 `null`。这正是第一个 `remove` 方法所需要的，但第二个 `remove` 方法必须返回一个布尔值。所以，在第二个方法中，我们必须将想删除的项与 `removeEntry` 的返回值进行比较，来得到所希望的布尔值。

学习问题 14 `remove` 的前一个定义中的 `return` 语句能写成下面这样吗？

- a. `return result.equals(anEntry);`
- b. `return result != null;`

学习问题 15 考虑包 `aBag`，它是类 `ArrayBag` 的一个实例。`ArrayBag` 中的数组 `bag` 含有 `aBag` 中的项。如果数组中含有字符串 "A", "A", "B", "A", "C"，为什么 `aBag.remove("B")` 将数组的内容改变为 "A", "A", "C", "A", `null`，而不是 "A", "A", "A", "C", `null`，或是 "A", "A", `null`, "A", "C"？

私有方法 `removeEntry` 的定义。现在回过头来看在段 2.26 中为从包中删除指定项而写的伪代码。私有方法 `removeEntry` 假定，项的查找已经完成，所以可以忽略伪代码中的第一步。不管怎样，伪代码的其他部分给出了删除一个项的基本逻辑。可以修改伪代码，如下所示。

```
// Removes and returns the entry at a given index within the array bag.  
// If no such entry exists, returns null.  
if(包不空且给定的下标不是负数)  
{  
    result = bag[givenIndex]  
    bag[givenIndex] = bag[numberOfEntries - 1]  
    bag[numberOfEntries - 1] = null  
    计数器numberOfEntries减1  
    return result  
}  
else  
    return null
```

前一段给出的方法 `remove` 的定义，将 `getIndexOf` 返回的整数传给了 `removeEntry`。因为 `getIndexOf` 可能返回 `-1`，故 `removeEntry` 也必须对这样的参数值进行查找。所以如果包不空——即如果 `numberOfEntries` 大于 `0`——且 `givenIndex` 大于等于 `0`，则

`removeEntry` 删除位于 `givenIndex` 的数组项，用最后一项来替换它，并让 `numberOfEntries` 减 1。然后方法返回删除的项。但是，如果包是空的，则方法返回 `null`。

方法的代码如下。

```
// Removes and returns the entry at a given index within the array bag.
// If no such entry exists, returns null.
// Preconditions: 0 <= givenIndex < numberOfEntries;
//                 checkIntegrity has been called.
private T removeEntry(int givenIndex)
{
    T result = null;
    if (!isEmpty() && (givenIndex >= 0))
    {
        result = bag[givenIndex];                      // Entry to remove
        bag[givenIndex] = bag[numberOfEntries - 1]; // Replace it with last entry
        bag[numberOfEntries - 1] = null;                // Remove last entry
        numberOfEntries--;
    } // end if
    return result;
} // end removeEntry
```

- 2.30 找到要删除的项。现在需要考虑如何找到要从包中删除的项，这样才可以将它的下标传给 `removeEntry`。方法 `contains` 执行的查找，与 `remove` 的定义中用来查找 `anEntry` 的机制是一样的。遗憾的是，`contains` 返回真或假；它不返回在数组中找到的项的下标。所以在定义这个方法时不能简单地调用那个方法。



设计决策：方法 `contains` 应该返回找到项的下标吗？

我们应该修改 `contains` 的定义，让它返回一个下标而不是一个布尔值吗？不应该。作为一个公有方法，`contains` 不应该提供给客户这样的实现细节。客户不应该期望包的项放在数组中，因为它们没有特定的次序。不应该改变 `contains` 的规范说明，而是应该遵循最初的规划，定义一个私有方法来查找一个项并返回它的下标。

- 2.31 `getIndexOf` 的定义。`getIndexOf` 的定义与 `contains` 的定义很像，我们回忆段 2.21 中它的循环是这样的：

```
boolean found = false;
int index = 0;
while (!found && (index < numberOfEntries))
{
    if (anEntry.equals(bag[index]))
    {
        found = true;
    }
    else
        index++;
} // end while
```

这个循环的结构适用于方法 `getIndexOf`，但当找到项时必须保存 `index` 的值。方法将返回这个下标而不是一个布尔值。

为修改前面这个循环，将其用在 `getIndexOf` 中，我们定义了一个整数变量 `where`，来记录当 `anEntry` 等于 `bag[index]` 时 `index` 的值。所以 `getIndexOf` 的定义是这样的：

```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
// Precondition: checkIntegrity has been called.
private int getIndexOf(T anEntry)
```

```

{
    int where = -1;
    boolean found = false;
    int index = 0;
    while (!found && (index < numberOfEntries))
    {
        if (anEntry.equals(bag[index]))
        {
            found = true;
            where = index;
        }
        else
            index++;
    } // end while
    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array
    return where;
} // end getIndexOf

```

方法 `getIndexOf` 返回 `where` 的值。注意到，我们将 `where` 初始化为 `-1`，这是没找到 `anEntry` 时返回的值。

 学习问题 16 就在方法 `getIndexOf` 的 `return` 语句之前，能添加什么 `assert` 语句，来表示方法能够返回的可能值？

学习问题 17 修改方法 `getIndexOf` 的定义，让它不使用布尔值。

旁白：正向思考

与方法 `contains` 不一样，方法 `getIndexOf` 将布尔变量 `found` 仅用来控制循环，而不是作为一个返回值。所以我们可以修改逻辑，以避免非操作符 `!` 的使用。

我们使用变量 `stillLooking` 来替代 `found`，将它初始化为真。然后可以将布尔表达式 `!found`，替换为 `stillLooking`，如你在下面方法 `getIndexOf` 的定义中所见：

```

// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
private int getIndexOf(T anEntry)
{
    int where = -1;
    boolean stillLooking = true;
    int index = 0;
    while (stillLooking && (index < numberOfEntries))
    {
        if (anEntry.equals(bag[index]))
        {
            stillLooking = false;
            where = index;
        }
        else
            index++;
    } // end while
    return where;
} // end getIndexOf

```

如果在数组中找到 `anEntry`，则 `stillLooking` 将置为假以结束循环。有些程序员倾向于正向思考，如这个版本中的这样，而另外一些人觉得 `!found` 就已经非常清楚了。

2.32 方法 `contains` 定义的修改。完成 `remove` 和它调用的私有方法的定义后，就知道方法 `contains` 可以调用私有方法 `getIndexOf`，得到比段 2.21 所给的更简单的定义。回忆一下，如果 `anEntry` 在包中，则表达式 `getIndexOf(anEntry)` 返回 $0 \sim \text{numberOfEntries}-1$ 之间的一个整数，否则返回 -1 。即如果 `anEntry` 在包中，则 `getIndexOf(anEntry)` 大于 -1 。所以可以定义 `contains` 如下：

```
public boolean contains(T anEntry)
{
    checkIntegrity();
    return getIndexOf(anEntry) > -1;
} // end contains
```

因为已经修改了 `contains` 的定义，所以应该再次测试它。为此，我们还要测试私有方法 `getIndexOf`。

 **注：**`contains` 方法和第二个 `remove` 方法都必须执行类似的对项的查找。将查找功能单独放在方法 `contains` 和 `remove` 都能调用的一个私有方法内，使得我们的代码更易调试及维护。这个策略等同于，将删除操作定义在两个 `remove` 方法都调用的私有方法 `removeEntry` 中时所使用的策略。

 **设计决策：**什么方法应该调用 `checkIntegrity`？

类 `ArrayBag` 的关键点是数组 `bag` 的分配。你已经看到，像 `add` 这样的依赖于这个数组的方法，都是先调用 `checkIntegrity`，以确保构造方法已经完全初始化了 `ArrayBag` 对象，包括数组的分配。我们可以坚持，在直接涉及数组 `bag` 的每个方法中调用 `checkIntegrity`，不过我们选择更加灵活的做法。例如，私有方法 `getIndexOf` 和 `removeEntry` 直接访问 `bag`，但它们不调用 `checkIntegrity`。为什么？删除给定项的 `remove` 方法调用 `getIndexOf` 和 `removeEntry`。如果这两个私有方法都调用了 `checkIntegrity`，则它被公有方法调用两次。所以，对于这个具体实现来说，我们在公有方法中调用 `checkIntegrity`，并为两个私有方法添加一个前置条件，来说明 `checkIntegrity` 必须要先调用。因为它们是私有方法，这样的前置条件只给我们这些实现者和维护者使用。一旦做了这个抉择，其他的 `remove` 方法和方法 `contains` 都必须调用 `checkIntegrity`，因为它们每一个都只调用这两个私有方法中的一个。

注意，私有方法 `getIndexOf` 和 `removeEntry` 都执行一个已定义好的任务。它们不再为第二个任务——检查初始化——负责。

 **程序设计技巧：**即使可能已经有了方法的正确定义，但如果想到了一个更好的实现，不要犹豫地去修改它。肯定要再次测试方法！

2.33 测试。我们的 `ArrayBag` 类基本上完成了。可以使用前面测试过 `remove` 和 `clear` 的测试方法——我们假定它们是正确的。从一个不满的包开始，在线程序 `ArrayBagDemo3` 删除包中的项直到它为空时为止。它还包括了从满包开始的类似的测试。最后，应该将前面的测试整合起来再次运行它们。本书在线网站的源代码中，测试程序是 `ArrayBagDemo`，完整的类是 `ArrayBag`。

使用变长数组实现 ADT 包

一个数组有固定的大小，在数组创建前，这个大小或者由程序员选择，或者由用户选择。定长数组像是一间教室。如果教室含有 40 把椅子但只有 30 名学生，则我们会浪费 10 把椅子。如果 40 名学生上课，则教室是满的，且不能再容纳其他任何人。类似地，如果没有用到数组中的所有位置，则浪费了空间。如果需要更多的，则运气不佳。

所以，使用定长数组实现 ADT 包，限制了包的大小。当数组满了，因此也是包满了，则后续的 add 方法的调用都返回假。有些应用可以使用有有限容量的包或其他的集合。但对于其他应用，我们需要集合的大小不受约束地变大。现在介绍一组项如何能想要多大就要多大——在计算机内存的限度内——但仍在一个数组内。

变长数组

策略。当教室满了，能容纳更多学生的一种办法是移到一间更大的教室里。用类似的方式，当数组满了，可以将它的内容移到一个更大的数组中。这个过程称为数组的变长 (resizing)。图 2-7 显示两个数组：一个是有 5 个连续内存位置的原数组，另一个数组——两倍于原数组大小——在计算机的另一块内存中。如果将数据从原来的小数组中拷贝到新的大数组的开头部分，得到的结果像是扩展了原数组一样。这种机制的唯一不足是新数组的名字：你想让它与原数组同名。马上就会看到如何完成这个工作。

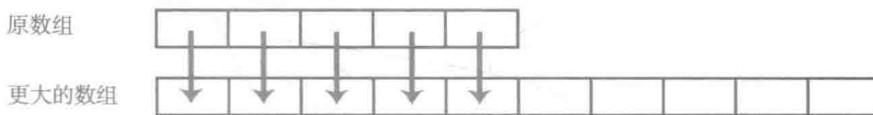


图 2-7 变长数组将它的内容拷贝到更大的第二个数组中

细节。假定已有 myArray 指向的数组，如图 2-8a 所示。我们先定义一个别名 oldArray，它也指向这个数组，如图 2-8b 所示。下一步是创建一个比原数组更大的新数组，让 myArray 指向这个新数组。如图 2-8c 所示，一般地，新数组要两倍于原数组的大小。最后一步是将原数组的内容拷贝到新数组中 (图 2-8d)，然后丢弃原数组 (图 2-8e)。下列伪代码概括了这些步骤：

```
oldArray = myArray
myArray = 其长度是 2 * oldArray.length 的新数组
将原数组 oldArray 中的项拷贝到新数组 myArray 中
oldArray = null // Discard old array
```



注：当数组不再被引用时，它占用的内存存在垃圾回收时被收回，就像是对其他对象的处理一样。

代码。将前一个伪代码转为 Java 时，使用 Java 类库中的方法 Arrays.copyOf (source-Array, newLength) 能帮忙做很多事情。例如，对如下的简单整数数组进行操作：

```
int[] myArray = {10, 20, 30, 40, 50};
```

此时，myArray 指向一个数组，如图 2-9a 所示。接下来，调用 Arrays.copyOf：

```
myArray = Arrays.copyOf(myArray, 2 * myArray.length);
```

2.34

2.35

2.36

2.37

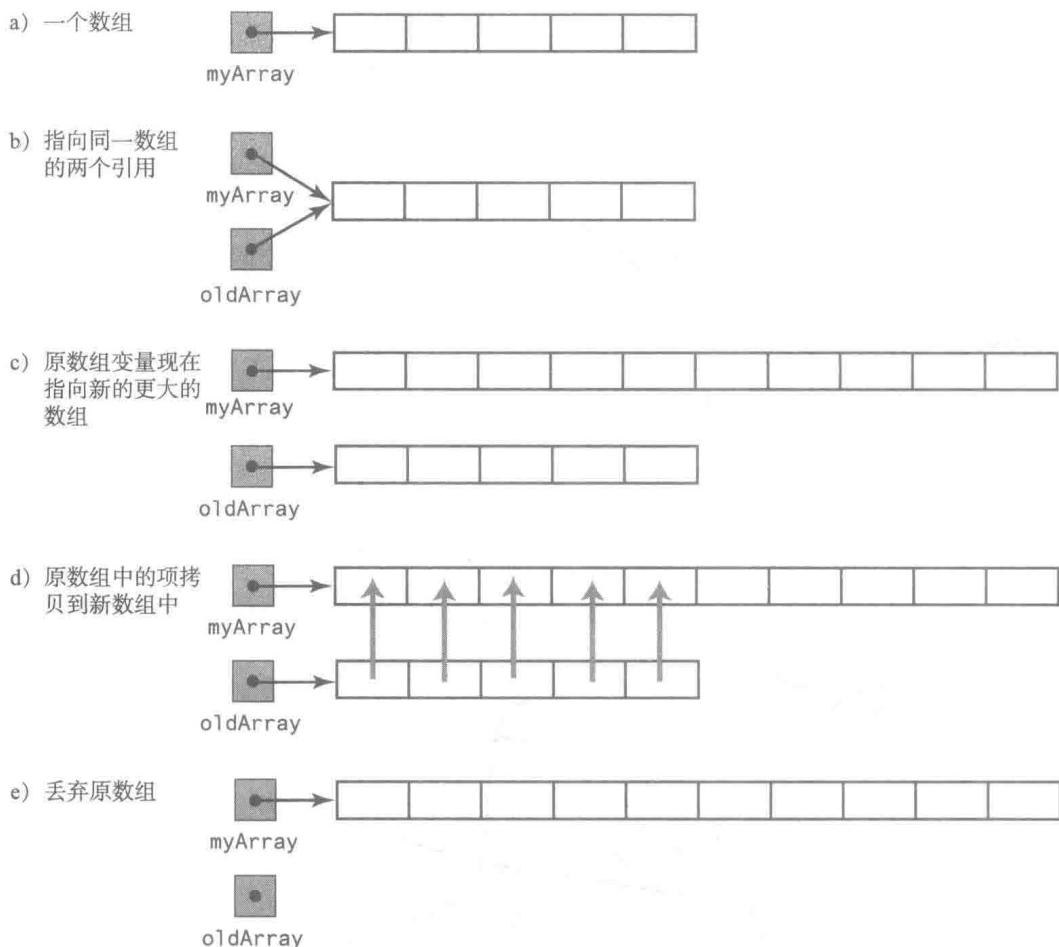


图 2-8 数组变长

变量 `myArray` 中的引用被赋给这个方法的第一个参数 `sourceArray`, 如图 2-9b 所示。接下来, 方法创建一个新的更大的数组, 并将参数数组中的项拷贝给它(图 2-9c)。最后, 方法返回指向新数组的一个引用(图 2-9d), 我们将这个引用赋给 `myArray`(图 2-9e)。

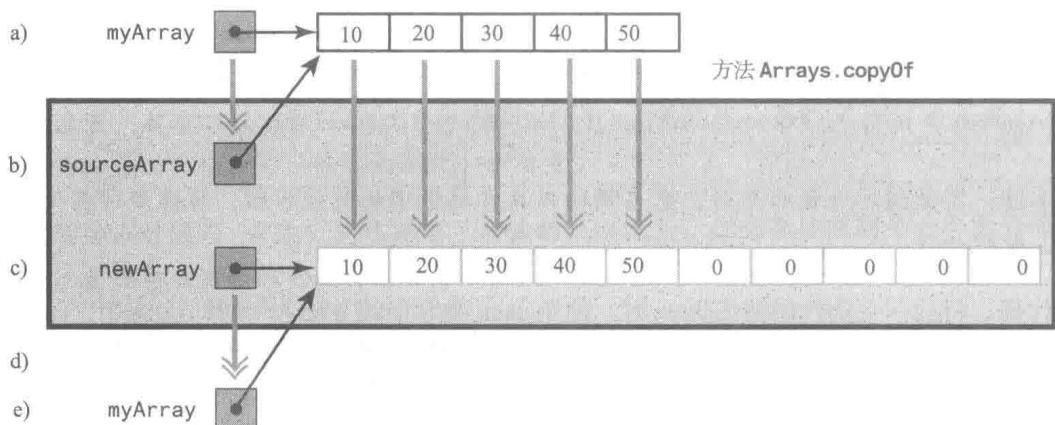


图 2-9 语句 `myArray = Arrays.copyOf(myArray, 2 * myArray.length);` 的效果
a) 参数数组; b) 指向参数数组的引用; c) 获得参数数组内容的新的更大的数组;
d) 指向新数组的返回值; e) 返回值被赋给参数变量



注：注意到，图 2-9 中的数组含有整数。这些整数是基本类型值，且依图这样占据数组中的位置。与之相对的，例如图 2-6 中的数组，保存的是指向对象的引用而不是对象本身。

数组变长或许没有第一眼看上去这样有吸引力。每次扩展数组大小时，必须拷贝它的内容。如果每次仅需一个额外空间而让数组增大一个元素，则这个过程将耗时过大。例如，如果含 50 个元素的数组满了，为了容纳另一个项，需要将数组拷贝到有 51 个元素的数组中。再添加一个项时又会要求你将含 51 个元素的数组拷贝到含 52 个元素的数组中，以此类推。每次添加都会导致数组的拷贝。如果在原含 50 个项的数组中添加 100 项，则要拷贝 100 次数组。2.38

另一种做法，将数组扩展 m 个元素，将拷贝开销分摊在 m 次添加上而不是集中在一次上。每次当数组满时倍增它的大小，这是一种典型的方法。

例如，当在含 50 个项的满数组中添加一个项时，在进行添加前先将 50 个元素的数组拷贝到 100 个元素的数组中。然后接下来的 49 次添加都可快速完成而不需要拷贝数组。所以数组拷贝只需一次。



程序设计技巧：当增大数组时，将它的项拷贝到更大的数组中。应该充分地扩展数组，以减少拷贝开销的影响。常用的办法是倍增数组大小。



注：“变长数组”的说法实际上是用词不当，因为数组的长度不会改变。变长数组的过程是创建了一个含有原数组项的全新的数组。给新数组与原数组一样的名字——换句话说，指向新数组的引用赋给了保存指向原数组引用的变量。然后丢弃原数组。



注：引入一个类

若一个类用到了 Java 类库中的类，则它的定义前必须有 `import` 语句。例如，要使用类 `Arrays`，应该将下面的语句写在你的类定义和描述性注释之前：

```
import java.util.Arrays;
```

有些程序员将这条语句中的 `Arrays` 替换为星号，为的是在他们的程序中可以使用包 `java.util` 中的所有类。



学习问题 18 考虑下列语句定义的字符串数组：

```
String[] text = {"cat", "dog", "bird", "snake"};
```

为数组 `text` 增大 5 个元素的容量且不改变当前内容的 Java 语句是什么？

学习问题 19 考虑字符串数组 `text`。如果放到这个数组中的字符串个数小于它的长度（容量），你如何减少数组长度而不改变它的当前内容？假定字符串个数保存在变量 `size` 中。

包的新实现

方法。可以通过变长数组来修改 ADT 包的前一个实现，这样包的容量仅由计算机可用的内存量来限定。如果查看程序清单 2-1 中 `ArrayBag` 类的框架，会明白我们需要修改什么。下面详细说明这些任务：

- 将类名修改为 `ResizableArrayBag`，这样我们就能区分两个实现了。

- 从数组 bag 的声明中删除标识符 final，以便它能变大小。
- 修改构造方法的名字以匹配新的类名。
- 修改方法 add 的定义，让它总能容纳新项。方法永远不会返回假，因为包永远不会满。类的其他部分将保持不变。修改方法 add 是这个列表中唯一的实质性工作。

2.40 方法 add。下面是方法 add 的定义，与段 2.15 结尾处的一样：

```
public boolean add(T newEntry)
{
    checkIntegrity();
    boolean result = true;
    if (isArrayFull())
    {
        result = false;
    }
    else
    { // Assertion: result is true here
        bag[numberOfEntries] = newEntry;
        numberofEntries++;
    } // end if
    return result;
} // end add
```

因为包永远不会满，故 add 应该永远返回真。为达到这个目标，当 isArrayFull 返回真时我们倍增数组 bag 的大小，而不是将 result 设置为假。为了变长数组，我们将定义并调用另一个私有方法 doubleCapacity，其规范说明如下：

```
// Doubles the size of the array bag.
private void doubleCapacity()
```

假定我们已经定义了这个私有方法，则可以修改 add 方法，如下所示：

```
/** Adds a new entry to this bag.
 * @param newEntry The object to be added as a new entry.
 * @return True. */
public boolean add(T newEntry)
{
    checkIntegrity();
    if (isArrayFull())
    {
        doubleCapacity();
    } // end if
    bag[numberOfEntries] = newEntry;
    numberofEntries++;
    return true;
} // end add
```

2.41 私有方法 doubleCapacity。使用之前在段 2.37 中描述过的技术来变长数组。因为我们增大了包的容量，所以必须检查新的容量不会超出 MAX_CAPACITY。在构造方法中也做同样的检查，但不是重复这段代码，而是定义让构造方法和 doubleCapacity 都能调用的另一个私有方法，来强制限制包的容量：

```
// Throws an exception if the client requests a capacity that is too large.
private void checkCapacity(int capacity)
{
    if (capacity > MAX_CAPACITY)
        throw new IllegalStateException("Attempt to create a bag whose " +
                                         "capacity exceeds allowed " +
                                         "maximum of " + MAX_CAPACITY);
} // end checkCapacity
```

方法 doubleCapacity 现在的定义如下：

```

// Doubles the size of the array bag.
// Precondition: checkIntegrity has been called.
private void doubleCapacity()
{
    int newLength = 2 * bag.length;
    checkCapacity(newLength);
    bag = Arrays.copyOf(bag, newLength);
} // end doubleCapacity

```

类 ResizableArrayBag。新的类可从本书的在线网站上获得。你应该研究那些细节。

2.42



设计决策：你可能想知道，我们在定义类 ResizableArrayBag 时，对如下的这些问题，是如何决策的：

- 为什么方法 add 是一个布尔方法而不是一个 void 方法？它永远返回真！
- 为什么我们要定义私有方法 doubleCapacity？只有 add 这一个方法调用它！

类实现了接口 BagInterface，所以当定义 add 时我们遵从它的规范说明。结果，有两种不同的实现，ArrayBag 和 ResizableArrayBag，同一个客户中这两种实现可能都会用到。对第二个问题的回答，反映了我们解决问题的方法。为实现 add，当数组满时首先要变长数组。不是在方法 add 内分神来执行这个任务，而是选择规范说明一个私有方法来扩展数组。不可否认，事实证明这个私有方法的定义很短。现在我们可以将私有方法的方法体集成到 add 中，但我们没有迫不得已的理由这样做。另外，保留私有方法，我们坚持的是一个方法应该执行一个动作的哲学理念。



学习问题 20 可以添加到类 ResizableArrayBag 中，使用给定数组的内容来初始化包的构造方法是什么？

学习问题 21 在前一个问题中描述的构造方法的定义中，有必要将参数数组中的项拷贝到数组 bag 中吗？或是一个简单的赋值语句 (bag = contents) 就足够了？

学习问题 22 使用数组来组织数据的好处是什么？坏处是什么？

测试类。测试 ResizableArrayBag 类的程序可以创建一个包，它的初始容量很小——例如 3。选择这个数能让我们很容易测试包增大其容量的能力。例如，当添加第 4 个项时，包的容量要倍增到 6。第 7 次添加时，容量再次倍增，这次是增到 12。可以从本书的在线网站上获得这个程序，程序名为 ResizableArrayBagDemo。

2.43



程序设计技巧：实现了声明 ADT 操作的单一接口的类，应该将定义在接口中的方法声明为自己的公有方法。但是，类还可以定义私有方法和保护方法。

使用数组实现 ADT 包的优缺点

本章讨论了 ADT 包使用数组保存包项的两种实现方式。数组易于使用，且如果知道任意的元素下标，则能对它立即进行访问。因为我们知道数组中最后一项的下标，所以删除它是简单且快速的。类似地，在数组尾添加一项也同样简单和快速。另一方面，删除某一项时，如果它位于其他两项的中间，则需要避免在数组中留有空隙。为此，用数组的最后一项来替换被删除的项。这增加一点点的执行时间，因为时间都花在了所需项的寻找上。本书后面我们将详细讨论这样的查找机制。

2.44

使用定长数组限制了包的容量，这通常是不利因素。变长数组能动态增大数组的大小，但需要拷贝数据。你应该知道，我们拷贝的数组项都是引用，所以不会占据太多空间，移动时也不会占用很多时间。Java之外的有些语言在数组中保存数据本身。这样的情形下，移动大量的复杂对象可能会相当耗时。



注：当使用数组来实现 ADT 包时，

- 向包中添加项是快速的。
- 删除未指定的项是快速的。
- 删除某个项时需要时间来找到这个项。
- 增大数组的大小需要时间来拷贝它的项。

本章小结

- 可以使用一个 Java 数组来实现 ADT 包，这相对简单，但其他的实现方式也是可能的。
- 在数组最后一项的后面添加一个项，不会影响已有项的位置。类似地，从数组中删除最后一项也不会影响已有项的位置。
- 因为包不维护项的次序，故删除一个项不需要将后续的所有数组项前移一个位置。相反，可以用数组中最后一项来替换你想删除的项，然后将最后一项替换为 `null`。
- 当你预料到要设计的类很长且复杂时，在其他方法之前标出并实现这个类的重要或核心方法，是一个好用的策略。对于其余的方法先使用存根。
- 在开发的每个阶段都要测试类，特别是在添加了重要的方法后。
- 使用定长数组可能导致包满了。
- 变长数组，看起来好像是数组能改变大小。为此，分配一个新数组，从原数组中将项拷贝到新数组中，使用原来的变量指向新数组。
- 变长数组能让你实现集合，其内容个数仅受计算机内存大小的限制。
- 你应该练习编写安全的程序。例如，ADT 包的实现中，在使用之前要检查包是否完全初始化，并检查它的容量不会超出给定的限度。

程序设计技巧

- 当定义一个类时，实现并测试一组核心方法。从向对象集合中添加对象的方法，或与实现方式关系最密切的方法入手。
- 方法不应该返回指向类中私有数据域的引用，而是应该返回指向数据域拷贝的引用。
- 不要等到完全实现 ADT 后才测试它。写存根，这是所需方法的不完整定义，可以尽早开始测试。
- 即使可能已经有了方法的正确定义，但如果你想到了一个更好的实现，不要犹豫地去修改它。肯定要再次测试方法！
- 当增大数组时，将它的项拷贝到更大的数组中。应该充分地扩展数组，以减少拷贝开销的影响。常用的办法是倍增数组大小。
- 实现了声明 ADT 操作的单一接口的类，应该将定义在接口中的方法声明为自己的公有方法。但是，类还可以定义私有方法和保护方法。

练习

1. 为什么类 `ArrayBag` 中的方法 `getIndexOf` 和 `removeEntry` 是私有方法而不是公有方法?
2. 为 ADT 包实现方法 `replace`, 用一个给定对象替换当前包中指定的对象, 并返回原对象。
3. 修改段 2.23 中给出的方法 `clear` 的定义, 让其更有效率, 且仅调用 `checkIntegrity` 方法。
4. 修改段 2.27 中给出的方法 `remove` 的定义, 让其从包中删除一个随机项。这个修改会影响到类 `ArrayBag` 中的其他方法吗?
5. 为类 `ArrayBag` 定义方法 `removeEvery`, 从包中删除给定项的所有出现。
6. 类 `ArrayBag` 的实例有固定大小, 而 `ResizableArrayBag` 的实例则不是。给出当包的大小是
 - a. 定长的
 - b. 变长的
 合适示例。
7. 假定想定义类 `PileOfBooks`, 实现第 1 章项目 2 中描述的接口。包是表示一摞书的合理集合吗? 解释之。
8. 考虑段 2.39 到段 2.43 中讨论的类 `ResizableArrayBag` 的实例 `myBag`。假定 `myBag` 的初始容量是 10。在
 - a. 向 `myBag` 中添加了 145 个项后
 - b. 向 `myBag` 中再添加 20 个项后
 数组 `bag` 的长度分别是多少?
9. 考虑接受类 `ArrayBag` 的实例作为参数的一个方法, 方法返回类 `ResizableArrayBag` 的一个实例, 实例中所含的项与参数所给的包的项相同。分别基于下列情况定义这个方法:
 - a. 在类 `ArrayBag` 内。
 - b. 在类 `ResizableArrayBag` 内。
 - c. 在类 `ArrayBag` 和 `ResizableArrayBag` 的客户内。
10. 假定包中含有 `Comparable` 对象, 例如字符串。一个 `Comparable` 对象属于实现了标准接口 `Comparable<T>` 的一个类, 所以有方法 `compareTo`。为类 `ArrayBag` 实现下列方法:
 - 返回包中最小对象的方法 `getMin`。
 - 返回包中最大对象的方法 `getMax`。
 - 删除并返回包中最小对象的方法 `removeMin`。
 - 删除并返回包中最大对象的方法 `removeMax`。
11. 假定包含有 `Comparable` 对象, 如前一个练习中所描述的那样。为类 `ArrayBag` 定义一个方法, 返回由小于某个给定项的项组成的新包。方法的头可以如下所示:


```
public BagInterface<T> getAllLessThan(Comparable<T> anObject)
```

 确保你的方法不会影响到原包的状态。
12. 为类 `ArrayBag` 定义 `equals` 方法, 当两个包的内容相同时返回真。注意到, 两个相等的包应含有相同个数的项, 每个项出现在每个包中的个数应相等。每个数组中项的次序是无关的。
13. 类 `ResizableArrayBag` 有一个数组, 当向包中添加对象时其大小在增大。修改这个类, 使得当从包中删除对象时, 它的数组还可以缩小。完成这个任务需要两个新的私有方法, 如下所示:
 - 第一个新方法检查是否应该减小数组的大小:

```
private boolean isTooBig()
```

 如果包中的项数小于数组大小的一半且数组的大小大于 20 时, 该方法返回真。
 - 第二个新方法创建一个新数组, 其大小是当前数组大小的 3/4, 然后将包中的对象拷贝到新数组中:

```
private void reduceArray()
```

实现这两个方法，然后使用它们来定义两个 `remove` 方法。

14. 考虑前一个练习中描述的两个私有方法。
 - a. 方法 `isTooBig` 需要数组的大小大于 20。如果落下这个要求可能会发生什么问题？
 - b. 方法 `reduceArray` 与方法 `doubleCapacity` 并不类似，它没有让数组的大小减小一半。如果数组的大小减到一半而不是 $3/4$ 时，可能会出现什么问题？
15. 为类 `ResizableArrayBag` 定义第 1 章练习 5 描述的方法 `union`。
16. 为类 `ResizableArrayBag` 定义第 1 章练习 6 描述的方法 `intersection`。
17. 为类 `ResizableArrayBag` 定义第 1 章练习 7 描述的方法 `difference`。
18. 写一个 Java 游戏程序。从梅花纸牌 Ace,Two,Three,……,Jack,Queen 和 King 中随机选择 6 张纸牌。将这 6 张牌放到包中。一次一个人，每个游戏玩家都来猜测包中有哪张牌。如果猜测正确，则可以从包中删除这张牌，并将它还到玩家手中。当包为空时，手中牌数最多的玩家获胜。

项目

1. 设计并实现单人猜谜游戏，选择 n 个 $1 \sim m$ 之间的随机整数，要求用户来猜它们。同一个整数可能被选中多次。例如，游戏可能选中 $1 \sim 10$ 之间的以下 4 个整数：4,6,1,6。用户和游戏之间的交互可能是：

输入你猜测的 $1 \sim 10$ 之间被选中的 4 个整数；

>1 2 3 4

你的猜测有 2 个是正确的，再猜。

>2 4 6 8

你的猜测有 2 个是正确的，再猜。

>1 4 6 6

正确！再玩一次？不。

再见！

设计作为 ADT 的游戏。使用包来保存游戏选择的整数。整数 m 和 n 由客户指定。

2. 定义表示一个集合的类 `ArrayList`，并实现第 1 章段 1.21 中描述的接口。在实现中使用类 `ResizableArrayBag`。然后写一个程序，充分展示你的实现。
3. 重复前一个项目，使用变长数组而不是使用类 `ResizableArrayBag`。
4. 定义类 `PileOfBooks`，实现第 1 章项目 2 中描述的接口。实现中使用变长数组。然后写一个程序，充分展示你的实现。
5. 定义类 `Ring`，实现第 1 章项目 3 描述的接口。实现中使用变长数组。然后写一个程序，充分展示你的实现。
6. 定义类 `LineOfCars`，实现第 1 章项目 8 描述的接口。实现时使用变长数组。然后写一个程序，充分展示你的实现。
7. 可以使用一个集合或一个包来创建拼写检查器。集合或包用作字典，且含有一组正确拼写的单词。要检查一个单词的拼写是否正确，可以看它是否含在字典中。使用这种方法创建拼写检查器用来检查外部文件中保存的单词。为简化任务，限制字典的规模。
8. 重做前一个创建拼写检查器的项目，不过将要检查拼写的单词放到包中。字典（含有拼写正确的单词的集合或包），及要检查的单词的包之间的差，是拼写错误的单词的包。
- 9.（游戏）考虑第 1 章项目 9 中设计的洞穴系统。使用 `ArrayBag` 的实例保存 `Cave` 对象，从而实现类 `CaveSystem`。写一个测试这个类的程序。
- 10.（财务）设计并实现类 `CreditCardAccount`，它是第 1 章项目 10b 中描述的 `FinancialAccount` 类的子类。初始化操作中使用客户提供的值来设置数据。
- 11.（电子商务）考虑第 1 章项目 11 中描述的购物车。使用 `ArrayBag` 的实例保存挑选的要购买的商品，从而实现类 `ShoppingCart`。写一个测试这个类的程序。

异常

先修章节：补充材料 1、附录 B、附录 C.

异常 (exception) 是方法执行期间发生的不常见的情况或事件，由此会中断程序的执行。有些异常表示代码中的错误。修正这些错误可以避免异常，且不需要再担心它们。事实上，最终的代码没有迹象表明可能会发生异常。况且，如果代码完全正确，异常就不会发生。

从另一方面来说，程序员可以在特定条件下有意让异常发生。事实上，写 Java 类库代码的程序员就是这样做的。如果你细读这个类库的文档，会看到某些方法执行期间可能发生的异常的名字。我们必须了解异常，这样才可以使用这些方法。当这样的异常发生时我们应该怎么办？是否应该在自己的程序中有意引发一个异常？如果是，如何来做？这是本插曲将回答的一些问题。当我们讨论 ADT 操作失败时，这些知识尤其重要。

基础

当在方法内发生异常时，方法创建一个异常对象，并将它传给 Java 运行时系统。我们说方法抛出 (throw) 了一个异常。被抛出的异常是发给程序其他部分的一个信号，表示某些意外的事情发生了。根据异常类的类型，以及作为对象的异常通过其方法告诉我们的信息，代码可以对其进行适当的响应处理。当发现并响应异常时，就是处理 (handle) 了异常。

异常属于不同的类，不过所有这些类都是标准类 `Throwable` 的后代。`Throwable` 在 Java 类库中，不需要 `import` 语句就可以使用。异常分为以下三组：

- 受检异常，它必须被处理
- 运行时异常，它不需要处理
- 错误，它不需要处理

受检异常 (checked exception) 是程序执行期间发生的严重事件的后果。例如，如果程序从磁盘读入数据，而系统找不到含有数据的文件，将会发生受检异常。这个异常所属类的类名是 `FileNotFoundException`。用户提供给程序的可能是一个错误的文件名。写得好的程序应该提前预见到这个事件，可能要求使用者再次输入文件名，以便能从中恢复正常。这个名字，与 Java 类库中所有异常类的名字一样，是用来描述异常原因的。通常的做法是使用类名来描述异常。例如，可能会说发生了一个 `FileNotFoundException` 异常。受检异常的所有类都是类 `Exception` 的子类，`Exception` 是 `Throwable` 的后代。



注：Java 类库中的受检异常

Java 类库中的下列类表示一些受检异常，你或许会遇到它们：

```
ClassNotFoundException  
FileNotFoundException  
IOException  
NoSuchMethodException  
WriteAbortedException
```

J2.1

J2.2

J2.3 运行时异常 (runtime exception) 通常是程序中逻辑错误的结果。例如，数组下标越界导致 `ArrayIndexOutOfBoundsException` 类的异常。被 0 除导致 `ArithmaticException` 异常。虽然可以添加代码来处理运行时异常，但通常只需要修正程序中的错误。运行时异常的所有类都是类 `RuntimeException` 的子类，后者是 `Exception` 的后代。



注：Java 类库中的运行时异常

Java 类库中的下列类表示一些运行时异常，你或许会遇到它们：

```
ArithmaticException
ArrayIndexOutOfBoundsException
ClassCastException
EmptyStackException
IllegalArgumentException
IllegalStateException
IndexOutOfBoundsException
NoSuchElementException
NullPointerException
UnsupportedOperationException
```

J2.4 错误 (error) 是标准类 `Error` 或其后代类的一个对象。将这样的类都称为错误类 (error class)。注意到 `Error` 是 `Throwable` 的后代。一般地，错误是指发生了不正确的情况，如运行时内存不足。如果程序用到的内存超出了限度，则必须修改程序以使内存的使用更有效率，改变配置让 Java 能访问更多的内存，或是为计算机购买更多的内存。这些情况都太严重了，一般程序很难处理。所以，即使处理错误是合法的，一般地也不需要处理它们。

J2.5 图 JI2-1 展示了一些异常和错误类的层次关系。运行时异常，比如 `ArithmaticException`，是 `RuntimeException` 的后代。受检异常，例如 `IOException`，是 `Exception` 的后代，但不是 `RuntimeException` 的后代。序言的段 P.9 中定义的断言错误，是类 `AssertionError` 的一个对象，`Error` 是 `AssertionError` 的父类。在第 7 章中讨论递归时，将提到栈溢出错误。这个错误属于类 `StackOverflowError`。`StackOverflowError` 和 `OutOfMemoryError` 都派生于抽象类 `VirtualMachineError`，`Error` 也是 `VirtualMachineError` 的父类。到目前为止，首要的一点是，我们要知道 `StackOverflowError`、`OutOfMemoryError` 和 `AssertionError` 的祖先类是 `Error` 类而不是 `Exception` 类，不过所有的异常和错误都派生于 `Throwable`。



注：异常层次

受检异常、运行时异常和错误的类——共同称为异常类 (exception class)——都是标准类 `Throwable` 的后代。运行时异常的所有类都派生于 `RuntimeException`，而它又派生于 `Exception`。受检异常是派生于 `Exception` 的类的对象，但它不是 `RuntimeException` 的后代。运行时异常和错误称为未检异常 (unchecked exception)。



注：很多异常类都在包 `java.lang` 中，所以不需要引入。但有些异常类在另外的包中，它们必须要引入。例如，当在程序中使用类 `IOException` 时，必须使用引入语句

```
import java.io.IOException;
```

我们会在补充材料 2 中遇到这个异常。

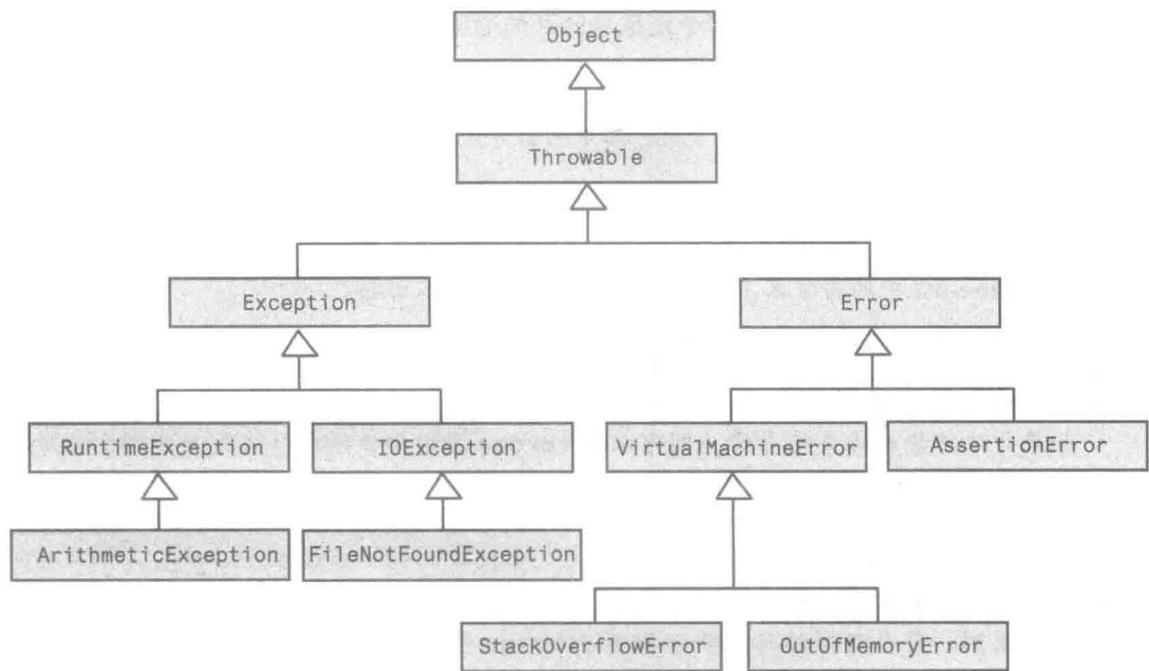


图 JI2-1 一些标准异常和错误类的层次关系

处理异常

当可能发生受检异常时，必须处理它。对于可能引发受检异常的方法，有两种选择：在方法内处理异常，或是告诉方法的客户来处理。

延缓处理：throws 子句

假定一个方法返回从磁盘读入的字符串。现在先不用担心这个方法如何完成这项任务，J2.6 我们将在补充材料 2 中学习如何写这样的程序。不过要知道，从磁盘读入时可能会发生错误。这件事有时会产生一个 `IOException` 异常。因为 `IOException` 是受检异常，所以它必须被处理。我们可以在方法体内处理异常。但有时，程序员不能肯定异常发生时怎样做对客户才是最好的。是应该终止执行，还是进行其他的处理更有意义？当不能肯定要采取哪个动作时，可以让方法的客户来处理异常。只要异常能在某个地方被处理，你就不需要在方法内来处理它。

一个可能导致受检异常但又不处理它的方法，就必须在方法头声明这件事。例如，如果方法 `readString` 可能抛出一个 `IOException` 异常但不处理它，则它的方法头应该是如下这样的：

```
public String readString(...) throws IOException
```

做标记的部分称为 `throws` 子句 (`throws` clause)。它表明方法 `readString` 不用负责处理执行期间可能发生的 `IOException` 类型的异常。但如果另一个方法调用 `readString` 方法，则那个方法必须处理异常。调用方法可以自己处理 `IOException`，也可以在它的方法头中包含一个 `throws` 子句，从而告诉它的客户来处理异常。最终，抛出的每个受检异常都应该在程序的某个地方被处理。

你可以在 `throws` 子句中列出多个用逗号分开的受检异常。



语法: `throws` 子句

方法头可以含有一个 `throws` 子句，其中列出方法可能抛出但不处理的异常。子句的语法如下：

```
throws exception-list
```

`exception-list` 中的异常名用逗号分隔。它们的次序不重要。



注：如果方法可能抛出一个受检异常，则或者在方法头写 `throws` 子句声明它，或者在方法内处理它。不这样做会导致语法错误。

如果方法可能抛出未检异常，则可以在 `throws` 子句中声明它，或是处理它，但也可以什么都不做。



注: javadoc 标签 `@throws`

对于方法可能抛出的每个异常，在方法头之前的 `javadoc` 注释中，应该用单独的一行来说明。每个这样的行都要以标签 `@throws` 开头，且它们应该按异常名的字典序排列。所有受检异常都必须被说明。

运行时异常可说明也可不说明，且一般不说明。但是，设计人员可以说明客户或许有理由很想处理的那些运行时异常。事实上，你会在 Java 类库中遇到一些被说明的运行时异常。但要知道，你使用的方法或许会导致一个未说明的运行时异常。如果你决定要说明运行时异常，则这些说明不应该依赖于方法的定义方式。所以，标出方法可能抛出的异常，应该作为设计及规范说明的部分，而不是实现的部分。



注：如果方法抛出一个异常但没有处理，则方法结束执行

如果方法抛出一个异常但没有处理它，则将结束方法的执行。但如果是客户处理异常，无论是方法的客户，还是客户的客户，程序都继续执行。对于一个受检异常，记住，如果方法不处理它，则必须在 `throws` 子句中声明它。



程序设计技巧：当定义一个可能抛出受检异常的方法时，如果不能提供对异常的合理响应，则要在方法头写一个 `throws` 子句将异常传给方法的客户。避免在 `throws` 子句中使用 `Exception`，因为这样做，不会给其他程序员提供关于调用方法的任何有用信息。相反，要尽可能地指明异常。

现在处理: `try-catch` 块

要处理异常，必须先标出可能导致异常的 Java 语句。还必须决定要寻找哪个异常。方法的文档及 `throws` 子句会告诉我们可能发生哪些受检异常。这就是我们要处理的异常。

处理异常的代码含有两段。第一段 `try` 块 (`try block`) 含有可能抛出异常的语句。第二段含有一个或多个 `catch` 块。每个 `catch` 块 (`catch block`) 含有处理或捕获 (`catch`) 某种类型异常的代码。所以，因为调用方法 `readString` 而处理 `IOException` 的代码可能有如下的形式：

```

try
{
    <其他的代码>
    anObject.readString(. . .); // Might throw an IOException
    <更多其他的代码>
}
catch (IOException e)
{
    <响应异常的代码，可能含有下面这行:>
    System.out.println(e.getMessage());
}

```

try 块中的语句的运行，与没有这个块时是一样的。如果没发生异常，则 try 块全部执行，然后执行 catch 块后面的语句。但如果在 try 块内发生了 IOException，则执行立即转到 catch 块。现在已经捕获了异常。

catch 块的语法类似于一个方法定义。标识符 e 称为 catch 块参数 (catch block parameter)；它表示 catch 块将处理的 IOException 的对象。虽然 catch 块不是方法定义，但在 try 块内抛出一个异常，就像是调用一个 catch 块，其中参数 e 表示一个实际的异常。

作为一个对象，每个异常都有存取方法 getMessage，它返回抛出异常时创建的描述字符串。通过显示这个字符串，可以告诉程序员所发生异常的性质。

执行完 catch 块后，执行它后面的语句。但如果问题是严重的，最好的响应是中断程序吗？catch 块可以调用 exit 方法来终止程序，如下所示：

```
System.exit(-1);
```

作为参数赋给 System.exit 的数 -1，表示程序的异常结束。因为我们遇到了一个严重问题，故我们有意中断程序。

 **注：**如果没有处理受检异常，或在 throws 子句中声明它，则编译程序会有提示。方法的有些异常可以在方法的定义中来处理，而有些可以在它的 throws 子句中声明。一般地，不处理或声明运行时（未检）异常，因为它们表示程序的一个错误。当抛出这样的异常时会中断程序的执行。

 **注：**其参数是 C 类型的 catch 块，可以捕获类 C 或 C 的任何后代类的异常。

多个 catch 块

一个 try 块中的语句，可能抛出不同类型异常中的任意一个。例如，假定段 J2.7 的 try 块中的代码能抛出多个类型的受检异常。在这个 try 块后的 catch 块能捕获 IOException 类的异常，及从 IOException 类派生的任意类的异常。要捕获其他类型的异常，可以在 try 块后写多个 catch 块。当抛出一个异常时，catch 块出现的次序是有意义的。执行进入其参数与异常的类型相匹配的第一个 catch 块——按照出现的次序。

不好的 catch 块次序。例如，下列 catch 块次序不好，因为用于 FileNotFoundException 的 catch 块永远不会被执行：

```

catch (IOException e)
{
    ...
}

```

J2.8

J2.9

J2.10

J2.11

```

catch (FileNotFoundException e)
{
    ...
}

```

按照这个次序，任何 I/O 异常都将被第一个 catch 块所捕获。因为 FileNotFoundException 派生于 IOException，所以 FileNotFoundException 异常是 IOException 异常的一种，将与第一个 catch 块的参数相匹配。幸运的是，编译程序可能会对这个次序给出警告信息。

J2.12 好的 catch 块次序。正确的次序是，将更具体的异常放在其祖先类的前面，如下所示：

```

catch (FileNotFoundException e)
{
    ...
}
catch (IOException e) // Handle all other IOExceptions
{
    ...
}

```

 程序设计技巧：因为受检异常和运行时异常的类都以 Exception 为祖先，故避免在 catch 块中使用 Exception。而是尽可能地捕获具体的异常，且先捕获最具体的。

 语法：try-catch 块的语法如下：

```

try
{
    <可能引发异常的语句>
}
catch (exceptionType e)
{
    <响应异常的代码，可能含有下面这行：>
    System.out.println(e.getMessage());
}
<其他可能的catch块>

```

 程序设计技巧：如果可能，避免嵌套的 try-catch 块

虽然在 try 块或 catch 块中再嵌套 try-catch 块是合法的，但应该尽可能地避免这样做。先看看能不能用不同的逻辑来组织代码以避免嵌套。如果不可以，将内层块移到一个新方法中，然后在外层块中调用这个新方法。

如果必须嵌套 try-catch 块，则可以遵循以下指南。当 catch 块出现在另一个 catch 块中时，它们必须使用不同的标识符表示各自的参数。如果计划在 try 块内嵌套 try-catch 块，若外层 catch 块更适合处理相关的异常，则可以忽略内层的 catch 块。这种情形中，内层 try 块抛出的异常被与外层 try 块对应的 catch 块捕获。

抛出异常

虽然处理异常的能力十分有用，但知道如何抛出异常也很重要。本节来看看如何抛出异常。应该仅当你不能使用合理的方式来解决不正常或意外事件的情形下，才在方法内抛出异常。

J2.13 throw 语句。方法执行 throw 语句则有意抛出一个异常。一般的形式是

```
throw exception_object;
```

程序员通常不是使用一条单独的语句来创建异常对象，而是在 throw 语句中创建对象，

如下面这个例子所示：

```
throw new IOException();
```

这个语句创建类 `IOException` 的一个新对象并抛出它。与应该尽可能地捕获具体异常一样，抛出异常时也应该尽可能地具体。

虽然可以调用异常类的默认构造方法，比如在前一个例子中所做的那样，不过我们也能提供带字符串参数的构造方法。得到的对象的数据域中将含有该字符串，且在处理异常的 `catch` 块中可以使用这个对象和这个字符串。然后 `catch` 块可使用异常的方法 `getMessage` 来获取这个字符串，如之前所见。默认构造方法为这个字符串提供的是默认值。

Java 语法：`throw` 语句有下列语法：

```
throw exception_object;
```

其中 `exception_object` 是异常类的一个实例，一般地通过调用类的下列两个构造方法之一来创建：

```
new class_name()
```

或

```
new class_name(message)
```

在捕获异常的代码段中，通过异常的方法 `getMessage`，可以使用默认构造方法提供的字符串，或是字符串 `message`。



设计决策：如果发生了不常见的情况，我该抛出异常吗？

- 如果可以通过合理的方式解决不常见的情况，则可能会使用判定语句而不是抛出一个异常。
- 如果对不正常情况的几种解决办法都是可行的，且你想让客户来选择，则应该抛出一个受检异常。
- 如果程序员因不正确地使用你的方法而使得代码出错了，则你可以抛出一个运行时异常。但是，不应该仅是简单地为了不让客户去处理它，就抛出一个运行时异常。



程序设计技巧：如果方法含有一个抛出异常的 `throw` 语句，则在方法头添加一个 `throws` 子句，而不是在方法体内捕获异常。一般地，抛出异常及捕获异常应该在不同的方法内。



程序设计技巧：不要弄混关键字 `throw` 和 `throws`

在方法头中用 Java 保留字 `throws` 来声明这个方法可能抛出的异常。在方法体内用保留字 `throw` 实际抛出一个异常。



设计决策：当在循环体内发生异常时，是应该将 `try-catch` 块放在循环内，还是将整个循环放在 `try` 块内？

这个问题没有明确的答案。有些程序员在最接近抛出异常的代码中处理它，即他们将 `try-catch` 块放到循环中。另一些程序员追求最整洁的代码，故将循环放到后接一个或多个 `catch` 块的 `try` 块中。当然，评判代码的样式是一个主观的事情。下面的客

观准则供你参考：

- 如果你想在异常发生后循环还是有效的，则将 try-catch 块放到循环体内。
- 如果你想在异常发生时结束循环，则将循环放到后接一个或多个 catch 块的 try 块中。

J2.14



示例。虽然刚刚给出的两种选择在运行时间上没有明显的差别，但结果是不同的。例如，考虑下面含有 try-catch 块的循环：

```
int i = 0;
while (i < 20)
{
    try
    {
        if (i == 10)
            throw new Exception();
        else if (i % 2 == 0)
            System.out.println(i + " is even.");
    }
    catch(Exception e)
    {
        System.out.println("Exception: " + i + " is too large.");
    }
    i++;
} // end while
```

输出是

```
0 is even.
2 is even.
4 is even.
6 is even.
8 is even.
Exception: 10 is too large.
12 is even.
14 is even.
16 is even.
18 is even.
```

假定我们将循环放到 try 中，如下所示：

```
int i = 0;
try
{
    while (i < 20)
    {
        if (i == 10)
            throw new Exception();
        else if (i % 2 == 0)
            System.out.println(i + " is even.");
        i++;
    } // end while
}
catch(Exception e)
{
    System.out.println("Exception: " + i + " is too large.");
}
```

现在的输出是

```
0 is even.
2 is even.
4 is even.
6 is even.
8 is even.
Exception: 10 is too large.
```

使用链式数据实现包

先修章节：第1章、第2章、Java插曲2

目标

学习完本章后，应该能够

- 描述数据的链式组织方式
- 描述如何在结点链表的开头添加新结点
- 描述如何删除结点链表的首结点
- 描述如何在结点链表中找到某个特定数据
- 使用结点链表实现ADT包
- 描述基于数组实现和链式实现ADT包的不同

使用数组实现ADT包既有优点也有不足，如你在第2章所见。数组有固定的大小，所以它可能会满，或有一些未用的元素。当它变满时将项移到更大的数组中，从而变长数组。虽然变长数组可以为包提供所需要的空间，但在每次扩展数组时必须移动数据。

本章介绍一种数据组织方式，仅当新项需要时才使用内存，删除一项后，将不再需要的内存返回给系统。通过链式数据，这个新的组织方式可以避免在添加或删除包项时移动数据。这些特征使得包的这种实现方法是基于数组的实现方式的重要替代。

链式数据

在第2章，我们用教室做比喻，描述了数据如何保存在数组中。现在，我们使用教室来展示数据的另一种组织方式。3.1

想象将一间空教室（房间L）分配给一门课程。所有可用的椅子都在走廊里。选了这门课的学生可以得到一把椅子，将它带到教室中，并坐在那里。教室可以容纳大厅里的所有椅子。

走廊里的每把椅子的背面都印有一个编号。这个编号称为地址（address），永远不会改变，且椅子分配给学生时也不用考虑这个编号。所以最终教室内含有的椅子的地址可能不是连续的。

现在假定，Jill是教室L内坐在刚好30把椅子上的30名学生之一。每张椅子上绑有一张白纸。当Jill进入教室时，我们在她椅子的纸上写上教室内另一张椅子的编号（地址）。例如，Jill椅子的纸上可能写的是20。如果她的椅子的编号是15，则我们可以说，15号椅子指向（reference）20号椅子，且15号椅子和20号椅子是链接（link）的。因为所有的椅子都以这种方式链接到另一把椅子，所以我们说它们组成了一个椅子链表（chain）。

图3-1说明了有5把椅子的一个链表。没有椅子指向链表中的第一把椅子，但老师知道这个椅子的号码是22。注意到，链表中最后一把椅子不指向其他的椅子，这把椅子的纸上是空白的。

椅子链表提供了椅子的次序。假定链表中的第一把椅子是最近到达的学生的。这个学生的椅子上写的是恰在他之前到达的那位学生的椅子编号。除一个人以外，每个人的椅子都指3.2

向恰在他之前到达的那位学生的椅子。例外的那个学生是最先到达的那个人，他坐在最后的一把椅子上。这个最后的椅子不指向其他的椅子。

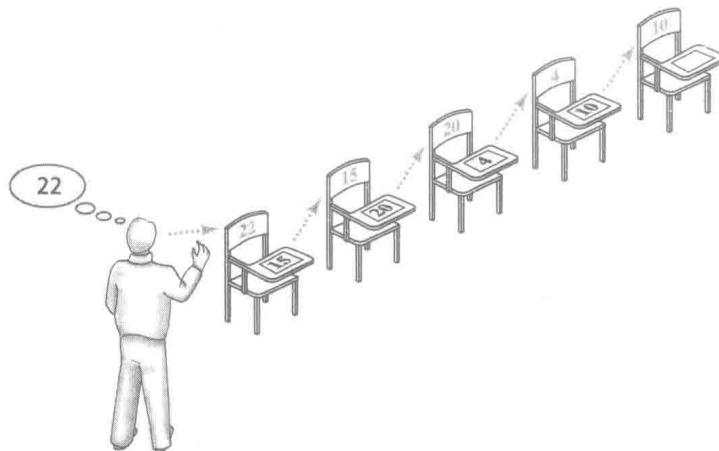


图 3-1 5 把椅子的链表

老师知道链表中第一把椅子的地址，所以可以向坐在第一把椅子上的同学问问题。然后，看写在第一把椅子的纸上的地址，或叫椅子编号，老师可以找到链表中的第二把椅子，并向坐在那里的人问问题。继续这种方式，老师可以按照椅子在链表中的次序依次访问 (visit) 每把椅子。最终老师到达链表中的最后一把椅子，它不指向其他椅子。注意到，老师找到这最后一把椅子上的学生的唯一方法，是从第一把椅子开始的。另外还注意到，老师只能以一种次序遍历 (traverse) 这个链表。在第 2 章类似的例子中，教室 A 中的老师可以按任意次序向任何一位学生提问。

添加到开头形成一个链表

3.3 一开始是如何形成椅子链表的？我们回到教室 L 空着而所有可用的椅子都在走廊中的那个时刻。

假定 Matt 最先到达。他从走廊里拿了一把椅子并走进教室。老师记下 Matt 的椅子编号 (地址)，在他椅子的纸上什么也不写，表示还没有其他学生到达。教室看起来像图 3-2 所示。

3.4 当第二位学生到达时，在新椅子的纸上写上 Matt 的椅子号，并让老师记下新椅子的编号。现在假定，老师每次只能记住一把椅子的编号。现在的教室如图 3-3 所示。新的椅子在链表的开头。

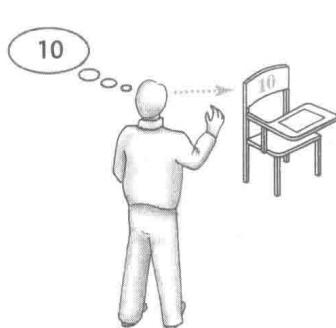


图 3-2 教室中有一把椅子

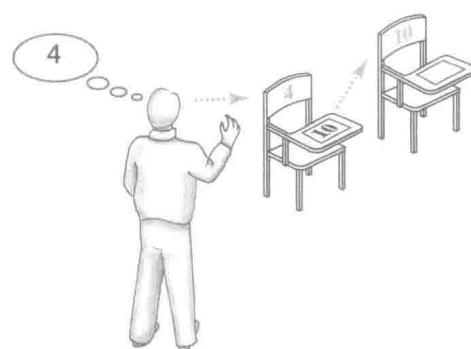


图 3-3 两把链起来的椅子，最新的椅子在第一个

当第三位学生到达时，我们在新椅子的纸上写下老师记住的椅子编号，那个是链表开头的那把椅子的编号。然后告诉老师记住新椅子的编号，现在它是链表的开头。现在教室的样子看起来如图 3-4 所示。

当所有的学生都到达后，老师仅知道最后到达的学生的椅子编号。在那位学生的椅子上写的是恰在他之前到达的学生的椅子编号。一般地，在每位学生的椅子上写下前一个到达的学生的椅子编号。因为 Matt 是最先到达的，所以在他的椅子的纸上仍是空白。在图 3-1 ~ 图 3-4 中，10 号椅子属于 Matt。

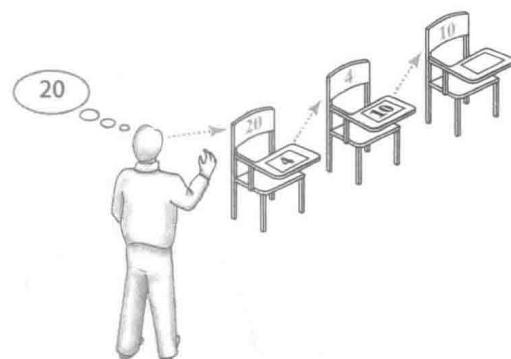


图 3-4 三把链起来的椅子，最新的椅子在第一个



学习问题 1 老师仅知道一把椅子的地址。

- 这把椅子在链表的什么位置：第一个、最后一个或是其他某个位置？
- 谁坐在那把椅子上：最先到达的学生、最后到达的学生或是其他人？

学习问题 2 椅子链表中，新的椅子添加在什么位置：开头、结尾或是其他位置？

下列伪代码详述了将新椅子添加在链表的开头从而组成一个椅子链表的步骤：

3.5

```
// Process the first student
newDesk 代表新学生的椅子
新学生坐在 newDesk 上
老师记下 newDesk 的地址

// Process the remaining students
while(有学生到达)
{
    newDesk 代表新学生的椅子
    新学生坐在 newDesk 上
    将老师记下的地址写在 newDesk 上
    老师记下 newDesk 的地址
}
```

ADT 包的链式实现

前一节描述了如何将数据链接在一起。本节用 Java 来表示这些思想，着手实现 ADT 包。

私有类 Node

从定义对象的类（称为结点（node））开始，它代表 Java 中的椅子。通常将结点链在一起形成一个数据结构。具体来说，我们的每个结点都有两个数据域：一个域指向一段数据——现在是包中的一个项——而另一个域指向另一个结点。包中的一个项类似于坐在椅子上的一个人。指向另一个结点的引用类似于写在每把椅子纸上的地址。

3.6

表示这些结点的类有下列形式：

```
class Node
{
    private T data; // Entry in bag
    private Node next; // Link to next node
    <构造方法>
```

```

    . . .
    <访问方法和赋值方法: getData, setData, getNextNode, setNextNode>
} // end Node

```

3.7 让我们集中讨论数据域。域 `data` 含有指向包中对象的一个引用。有时我们称这个域为结点的数据部分 (data portion)。这里, `data` 的数据类型表示为泛型 `T`。稍后就会看到 `T` 与将声明的包的类是同一个泛型。

域 `next` 中含有指向另一个结点的引用。注意到, 它的数据类型是 `Node`, 这是我们现在正在定义的类! 这样的循环定义可能会让你吃惊, 但在 Java 中这是完全合法的。这能让一个结点指向另一个结点, 就像我们的例子中一把椅子指向另一把椅子一样。注意到, 椅子不是指向另一把椅子中的学生。类似地, 结点也不是指向另一个结点中的数据, 而是指向另外一个完整的结点。有时我们称域 `next` 为结点的链接部分 (link portion)。图 3-5 说明了链接的两个结点, 它们都含有指向包中对象的引用。

3.8 类 `Node` 定义中的其他部分就没什么可说的了。用

来初始化结点的构造方法是有用的, 且因为数据域是私有的, 所以还要提供访问并修改其内容的方法。但它们真的是必需的吗? 如果想让 `Node` 像其他的类一样公用, 则这样的方法是必需的; 但是 `Node` 是 ADT 包的实现细节, 就应该对包的客户隐藏。将 `Node` 隐藏的一种方法是, 将它定义在包中, 且含在实现包的类中。另一种方法——此处我们所采用的——是在实现包的外部类 (outer class) 内定义 `Node`。因为它的位置含在另一个类内, 所以 `Node` 是一个内部类 (inner class)。我们将它声明为私有的。外部类可以按名直接访问内部类的数据域, 而不需要通过访问方法 (accessor) 和赋值方法 (mutator)。由此, 程序清单 3-1 中给出了更简单的 `Node` 定义。

程序清单 3-1 私有内部类 Node

```

1  private class Node
2  {
3      private T data; // Entry in bag
4      private Node next; // Link to next node
5
6      private Node(T dataPortion)
7      {
8          this(dataPortion, null);
9      } // end constructor
10
11     private Node(T dataPortion, Node nextNode)
12     {
13         data = dataPortion;
14         next = nextNode;
15     } // end constructor
16 } // end Node

```

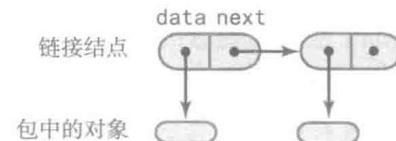


图 3-5 两个链接结点, 每个都指向一个对象数据

我们没有包含默认的构造方法, 因为我们不需要它。

因为 `Node` 是一个内部类, 故泛型 `T` 将与包含 `Node` 的外部类声明的泛型是一样的。所以, 我们没有在 `Node` 后写 `<T>`。但如果 `Node` 不是一个内部类, 而是有包访问或公有访问权限的, 则应该写 `Node<T>`。那种情况下, `Node` 还需要设置方法 (`set`) 和获取方法 (`get`) 用来访问它的数据域。

注：术语

嵌套类 (nested class) 是完全定义在另一个类定义之内的类。嵌套类可以是静态的，不过本书中并没有遇到这样的类。内部类是一个嵌套类但不是静态的。外部类或称包围类 (enclosing class) 含有一个嵌套类。顶层类 (top-level class) 或最外层类 (outermost class) 不是嵌套类。

类 LinkedBag 的框架

针对 ADT 包的这种实现方式，我们使用结点链表来保存包中的项。在前面教室示例中，老师记住了椅子链中第一把椅子的地址。类似地，我们的实现必须“记住”结点链表中第一个结点的地址。使用称为头引用 (head reference) 的数据域保存指向首结点的引用。第二个数据域可以记录包中项的个数，即链表中结点的个数。

实现 ADT 包的类 `LinkedBag` 的框架列在程序清单 3-2 中，其中还列出了作为内部类的 `Node`。回忆第 1 章程序清单 1-1 中介绍的接口 `BagInterface`。接口和实现接口的类为包中的对象定义了泛型。用来表示这个泛型的标识符 `T` 必须与用在内部类 `Node` 中的相一致。

程序清单 3-2 类 `LinkedBag` 的框架

```

1  /**
2   * A class of bags whose entries are stored in a chain of linked nodes.
3   * The bag is never full.
4  */
5  public final class LinkedBag<T> implements BagInterface<T>
6  {
7      private Node firstNode;           // Reference to first node
8      private int    numberOfEntries;
9
10     public LinkedBag()
11     {
12         firstNode = null;
13         numberOfEntries = 0;
14     } // end default constructor
15
16     <Implementation of the public methods declared in BagInterface go here. >
17
18     . . .
19
20     private class Node // Private inner class
21     {
22         <See Listing 3-1.>
23     } // end Node
24 } // end LinkedBag

```

数据域 `firstNode` 是结点链表的头引用。就像老师知道椅子链中第一把椅子的地址一样，`firstNode` 指向结点链表中的首结点。另一个数据域 `numberOfEntries` 记录当前包中项的个数。这个个数也是链表中结点的个数。初始时，包是空的，所以默认构造方法只需简单地将数据域 `firstNode` 初始化为 `null`，将 `numberOfEntries` 初始化为 0。

注意到，我们没有像在第 2 章中为类 `ArrayBag` 所做的那样，定义一个布尔域 `integrityOK`。本章稍后我们来解释原因。

注：支持类

类 `LinkedBag` 将它的数据——包项——保存在另一个类 `Node` 的实例中。我们喜欢将

`Node` 看作一个支持类 (supporting class)。作为一个内部类，它对 `LinkedBag` 的客户是隐藏的，但也是 `LinkedBag` 所属包的一部分。在后面的章节中我们将使用支持类，给它们赋予更多的职责。

定义一些核心方法

如前一章所述，写一个类时实现并测试一组核心方法常常是有利的。在实现像包这样的集合类时，将项添加到集合中的方法常常属于核心方法。另外，为验证向集合的添加是否正确，还需要查看集合项的方法。可用方法 `toArray` 来做这件事，而且它也是一个核心方法。前一章实现 `ArrayBag` 类的情形就是这样处理的，对于现在的类 `LinkedBag`，还是这样做。在定义其他方法之前，先来定义包的 `add` 方法和 `toArray` 方法。

3.10 方法 `add`：初建结点链表。 在段 3.3 中，当第一位学生到达时教室是空的。如我们在段 3.5 所说明的，采用下列步骤开始建立椅子链：

`newDesk` 表示新学生的椅子

新学生坐在`newDesk`上

老师记下`newDesk`的地址

下面是 `add` 方法用来将第一个项添加到初始为空的包中所采取的类似步骤。注意到，前面伪代码中的椅子类似于 `LinkedBag` 中定义的结点，学生类似于包的项——即结点中的数据——老师类似于 `firstNode`。

`newNode` 指向一个新的 `Node` 实例

将数据放到`newNode`中

`firstNode = newNode` 的地址

所以，当方法 `add` 将第一个项添加到初始为空的包中时，它创建一个新结点并将它变为单结点链表。

在 Java 中，这些步骤如下所示，其中 `newEntry` 指向要被添加到包中的项：

```
Node newNode = new Node(newEntry);
firstNode = newNode;
```

图 3-6 说明了这两步。图 3-6a 显示空链表及由第一条语句创建的结点。图 3-6b 显示第二条语句的结果。注意到，在图 3-6b 中，`firstNode` 和 `newNode` 都指向同一个结点。新结点插入完成后，应该只有 `firstNode` 指向它。我们可以将 `newNode` 设置为 `null`，但你马上就会看到，`newNode` 是方法 `add` 的局部变量。所以，在 `add` 结束执行后，`newNode` 不再存在。参数 `newEntry` 也是如此，它的行为像是一个局部变量。

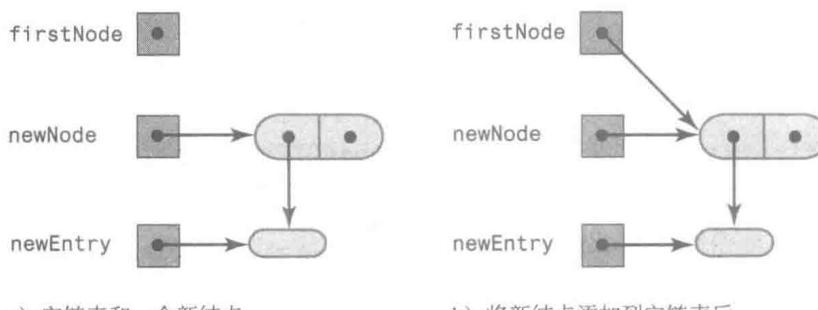


图 3-6 向空链表添加一个新结点

方法 add：添加到结点链表中。就像段 3.5 中将新椅子添加到已有链的开头一样，方法 add 将新结点添加到它自己链表的开头。在教室椅子的场景中，所需的步骤是：

newDesk 表示新学生的椅子

新学生坐在 newDesk 上

将老师记住的地址写到 newDesk 上

老师记下 newDesk 的地址

这些步骤的结果是，新椅子指向当前链中的第一把椅子且成为新的第一把椅子。

下面是 add 要采取的类似步骤：

newNode 指向一个新的 Node 实例

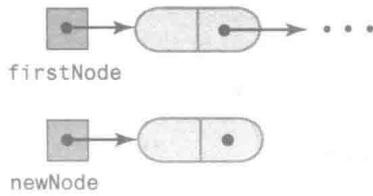
将数据放到 newNode 中

设置 newNode 的链接部分为 firstNode

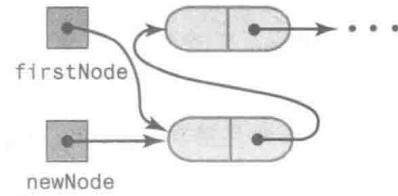
将 newNode 赋给 firstNode

即让新结点指向链表中的首结点，让它成为新的首结点。图 3-7 图示了这些步骤，由下列 Java 语句来实现：

```
Node newNode = new Node(newEntry);
newNode.next = firstNode;
firstNode = newNode;
```



a) 将结点添加在链表头之前的链表



b) 将结点添加在链表头之后的链表

图 3-7 将结点添加在链表头之前及之后的链表

如图 3-6 所述的将一个结点添加到空链表中，实际上与将结点添加到链表头是一样的。学习问题 3 要求你考虑这件事。



学习问题 3 上面“方法 add：初建结点链表”部分开发的将一个结点添加到空链表中的代码是

```
Node newNode = new Node(newEntry);
firstNode = newNode;
```

我们刚刚写的在链表头添加的代码是

```
Node newNode = new Node(newEntry);
newNode.next = firstNode;
firstNode = newNode;
```

为什么当链表为空时这三条语句也是正确的？

方法 add。正如你所见，当向包中添加新项时，出现空包似乎是一个特例，但实际上并不是这样的。方法 add 的定义用到了这个结论：

```
/** Adds a new entry to this bag.
 * @param newEntry The object to be added as a new entry.
 * @return True. */
public boolean add(T newEntry) // OutOfMemoryError possible
{
```

```

    // Add to beginning of chain:
    Node newNode = new Node(newEntry);
    newNode.next = firstNode;      // Make new node reference rest of chain
                                   // (firstNode is null if chain is empty)
    firstNode = newNode;          // New node is at beginning of chain
    numberEntries++;
    return true;
} // end add

```

3.13 内存不足错误。对于链式实现，包不会满。任何时候添加新项时，都可以为那个项创建一个新结点。所以方法 `add` 总返回真。但是，你的程序可能用光了计算机的所有内存。如果发生这种情形，再为新结点申请内存时将导致 `OutOfMemoryError` 错误。可以将这个条件解释为满包，但很少有客户能从这个错误中恢复。



注：分配内存

当使用 `new` 运算符时，将创建或称实例化一个对象。此时 Java 运行时环境为对象分配 (allocates) 或指派内存。当为链表创建结点时，有时称分配了结点。



安全说明：内部类 Node 应该执行安全检查吗？

因为 `Node` 是私有内部类，故我们将它看作外部类 `LinkedBag` 的实现细节。因此，让 `LinkedBag` 负责所有的安全检查。另外注意到，`Node` 的构造方法只做了简单的赋值操作，并没有抛出异常。即使不是这种情形，且 `Node` 可能抛出了异常，则 `LinkedBag` 也应该能处理它。



安全说明：类 LinkedBag 应该执行安全检查吗？

在段 3.9 的结尾处我们提到，`LinkedBag` 不需要布尔数据域 `integrityOK` 来检查构造方法是否完全执行。如你在程序清单 3-2 中所见，`LinkedBag` 的默认构造方法只进行了两个简单的赋值。实际上，所赋的这些值与省略构造方法时使用默认值赋给的值是一样的。这些赋值不会失败。

方法 `add` 分配了一个新结点。正如我们提到的，如果没有足够内存可用，这个分配就会失败。如果发生了 `OutOfMemoryError`，链表完好无损且保持不变。它或者是空的，或者含有之前调用 `add` 时已经赋值的那些结点。如果客户捕获了 `OutOfMemoryError` 并对包进行了处理，则这样的操作应属恰当。

因为任何 `LinkedBag` 对象的完整性已得到了维护，故此处不需要为类 `ArrayBag` 所添加过的那些安全检查。

3.14

方法 `toArray`。方法 `toArray` 返回当前在包中项的数组。实现了这个方法，就能在完成类 `LinkedBag` 的其他方法之前，先测试 `add` 方法是否正确工作。为访问包的项，我们需要从第一个结点开始访问链表中的每个结点。这个动作称为遍历 (traversal)，它类似于段 3.2 中所描述的访问椅子链中的每把椅子。

数据域 `firstNode` 中含有指向链表中首结点的引用。那个结点中含有指向链表中第二个结点的引用，第二个结点中含有指向第三个结点的引用，以此类推。为遍历链表，方法 `toArray` 需要一个临时的、局部变量 `currentNode`，依次指向每个结点。当 `currentNode` 指向我们想访问的数据的结点时，这个数据就在 `currentNode.data` 中。

初始时，想让 `currentNode` 指向链表的首结点，所以将它置为 `firstNode`。访问

`currentNode.data` 中的数据后，执行

```
currentNode = currentNode.next;
```

移到下一个结点。

再次访问 `currentNode.data` 中的数据，然后执行

```
currentNode = currentNode.next;
```

再次移动到下一个结点。继续这种方式，直到到达最后结点，而 `currentNode` 变为 `null` 时为止。

下列 `toArray` 方法用的正是这个思路：

```
/** Retrieves all entries that are in this bag.  
 * @return A newly allocated array of all the entries in the bag. */  
public T[] toArray()  
{  
    // The cast is safe because the new array contains null entries  
    @SuppressWarnings("unchecked")  
    T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast  
    int index = 0;  
    Node currentNode = firstNode;  
    while ((index < numberOfEntries) && (currentNode != null))  
    {  
        result[index] = currentNode.data;  
        index++;  
        currentNode = currentNode.next;  
    } // end while  
    return result;  
} // end toArray
```

 **程序设计技巧：**如果 `ref` 是指向链表中结点的引用，在使用它访问 `ref.data` 或是 `ref.next` 之前，要确定 `ref` 的值不是 `null`。另外，如果 `ref` 是 `null`，将发生 `NullPointerException`。

 **学习问题 4** 在前面 `toArray` 的定义中，`while` 语句使用逻辑表达式 `(index < numberOfEntries) && (currentNode != null)` 来控制循环。有必要对 `index` 和 `currentNode` 这两个值都测试吗？解释你的答案。

测试核心方法

之前我们知道，`add` 方法是类的基础，所以它是我们要实现及测试的核心方法之一。3.15方法 `toArray` 能让我们查看 `add` 是否正确工作，所以它也在核心组内。但是不在核心组内的方法有哪些呢？因为 `LinkedBag` 实现了接口 `BagInterface`，所以它必须定义这个接口内的每个方法。如前一章所述，我们要为接口内声明的但不在核心组内的这些方法写存根。因为方法 `getCurrentSize` 和 `isEmpty` 的定义很简单，所以我们在类 `LinkedBag` 中直接写初稿而不是写存根。

除了名字和用来创建包的类外，`LinkedBag`^② 的测试程序类似于前一章程序清单 2-2 中给出的 `ArrayBag` 的测试程序，有一个明显的区别是：虽然 `ArrayBag` 的实例可以变满，

② 类 `LinkedBag` 的这个版本可在本书的在线网站获得，名为 `LinkedBag1`。

但 `LinkedBag` 的实例却不会。程序清单 3-3 列出了这样一个测试程序的框架。注意到，这里的私有静态方法与第 2 章程序清单 2-2 中给出的一模一样。这是合理的，因为方法用 `BagInterface` 作为包的数据类型。

程序清单 3-3 测试类 `LinkedBag` 中某些方法的程序示例

```

1  /** A test of the methods add, toArray, isEmpty, and getCurrentSize,
2   *  as defined in the first draft of the class LinkedBag.
3  */
4  public class LinkedBagDemo1
5  {
6      public static void main(String[] args)
7      {
8          System.out.println("Creating an empty bag.");
9          BagInterface<String> aBag = new LinkedBag<>();
10         testIsEmpty(aBag, true);
11         displayBag(aBag);
12
13         String[] contentsOfBag = {"A", "D", "B", "A", "C", "A", "D"};
14         testAdd(aBag, contentsOfBag);
15         testIsEmpty(aBag, false);
16     } // end main
17
18     // Tests the method isEmpty.
19     // Precondition: If the bag is empty, the parameter empty should be true;
20     // otherwise, it should be false.
21     private static void testIsEmpty(BagInterface<String> bag, boolean empty)
22     {
23         System.out.print("\nTesting isEmpty with ");
24         if (empty)
25             System.out.println("an empty bag:");
26         else
27             System.out.println("a bag that is not empty:");
28
29         System.out.print("isEmpty finds the bag ");
30         if (empty && bag.isEmpty())
31             System.out.println("empty: OK.");
32         else if (!empty)
33             System.out.println("not empty, but it is: ERROR.");
34         else if (!empty && bag.isEmpty())
35             System.out.println("empty, but it is not empty: ERROR.");
36         else
37             System.out.println("not empty: OK.");
38     } // end testIsEmpty
39     <The static methods testAdd and displayBag from Listing 2-2 are here. >
40 } // end LinkedBagDemo1

```

方法 `getFrequencyOf`

3.16 为统计所给项在包中出现的次数，必须遍历结点链表，查看每个结点中的项。遍历非常类似于方法 `toArray` 中使用的方式。所以，如果让 `currentNode` 指向待查结点，则它的初值置为 `firstNode`——链表中的首结点——然后使用语句

```
currentNode = currentNode.next;
```

前移到下一个结点。使用这个方法，可以写如下这样的循环：

```
int loopCounter = 0;
Node currentNode = firstNode;
while ((loopCounter < numberOfEntries) && (currentNode != null))
```

```

{
    ...
    loopCounter++;
    currentNode = currentNode.next;
} // end while

```

虽然因为方法 `toArray` 要处理数组而使用变量 `index`, 但这里我们使用变量 `loopCounter`, 因为我们没有数组。你应该注意到, `loopCounter` 对结点进行计数以便控制循环; 它不对给定项在包中的出现次数进行计数。另外, 可以完全忽略 `loopCounter`, 但我们保留它, 用来进行逻辑检查。

在循环体内, 访问当前结点中的数据, 将它与作为参数传给方法的项进行比较。每次发现一对匹配的, 将次数计数加 1。所以, 方法 `getFrequencyOf` 的定义如下:

```

/** Counts the number of times a given entry appears in this bag.
 * @param anEntry The entry to be counted.
 * @return The number of times anEntry appears in the bag. */
public int getFrequencyOf(T anEntry)
{
    int frequency = 0;
    int loopCounter = 0;
    Node currentNode = firstNode;
    while ((loopCounter < numberOfEntries) && (currentNode != null))
    {
        if (anEntry.equals(currentNode.data))
            frequency++;
        loopCounter++;
        currentNode = currentNode.next;
    } // end while
    return frequency;
} // end getFrequencyOf

```

方法 `contains`

前一章——使用数组来表示包项——中, 通过检查每个数组元素——从下标 0 开始——来判定一个包中是否含有给定项, 直到或者找到所需的项或者发现它不在数组中。这里我们用类似的方法在一个链表中查找某个特定的数据, 每次查看链表中的一个结点。从首结点开始, 如果它不含有我们要找的项, 则查找第二个结点, 以此类推。3.17

当查找数组时, 我们使用下标。要查找链表, 我们使用指向结点的引用。所以, 就像在方法 `getFrequencyOf` 中一样, 使用一个局部变量 `currentNode` 指向想要检查的结点。初始时, 将 `currentNode` 设置为 `firstNode`, 然后在遍历链表时设置为 `currentNode.next`。但是, 不是像 `getFrequencyOf` 那样遍历整个链表, 而是当找到所需的项, 或是 `currentNode` 变为 `null`——此时项不在包中——时, 循环结束。

故方法 `contains` 的实现如下所示。

```

public boolean contains(T anEntry)
{
    boolean found = false;
    Node currentNode = firstNode;
    while (!found && (currentNode != null))
    {
        if (anEntry.equals(currentNode.data))
            found = true;
        else
            currentNode = currentNode.next;
    }
}

```

```

    } // end while
    return found;
} // end contains

```



学习问题 5 如果前面的 `contains` 方法中，`currentNode` 变为 `null`，当包非空时方法返回什么值？

学习问题 6 当包为空时，跟踪方法 `contains` 的执行，结果是什么？

从链表中删除一项

3.18 本章前面使用教室来描述如何形成一个数据链。可用的椅子放在教室外的走廊中。每把椅子的背面印有一个编号（地址），且在椅子上有一张白纸。当学生进入教室时，他们从大厅拿一把椅子。在新椅子的白纸上写下已在教室中的另一把椅子的编号，将新椅子的编号给老师。这种方式下，椅子一个链着一个，组成椅子链。如图 3-1 所见，没有椅子指向链中的第一把椅子，但老师知道它的地址。最后一把椅子不指向其他的椅子，它的纸上是空的。

离开教室（教室 L）的学生将他们的椅子放回走廊。这样的椅子可以重新分配给进入教室 L 的其他学生，或是共享这个走廊的其他教室的学生。假定你是教室 L 中的一名学生，但你想逃课。如果只简单地将椅子放回走廊，实际上并没有将自己从教室的椅子链中移走：另一把椅子或是老师仍指向你的椅子。我们需要你的椅子从链中断开。如何做到这一步，依赖于你椅子在链中的位置。可能有下面的情形：

- 情形 1：你的椅子在椅子链的第一个。
- 情形 2：你的椅子不在椅子链的第一个。

3.19 情形 1。图 3-8 说明的是情形 1，将第一把椅子从链中删除前的情形。下面是删除第一把椅子所必需的步骤：

- 1) 向老师要地址，找到第一把椅子。
- 2) 将写在第一把椅子上的地址给老师。这是链中第二把椅子的地址。
- 3) 将第一把椅子放回走廊。

图 3-9 所示为执行前两步之后的链。注意到，第一把椅子不再属于链中。技术上，它仍指向第二把椅子。但如果这把椅子被再次使用，则在它的纸上将会写新的地址。

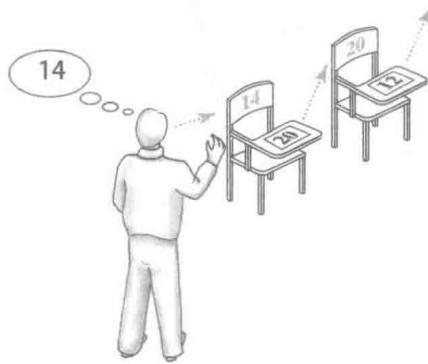


图 3-8 删除第一把椅子之前的椅子链

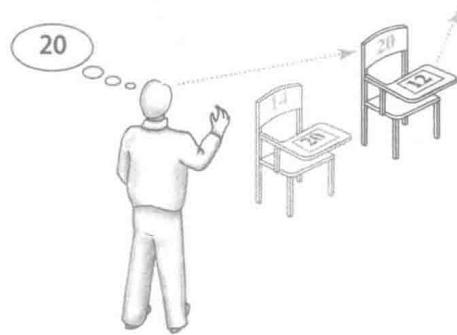


图 3-9 刚刚删除了第一把椅子之后的椅子链

3.20 情形 2。回忆一下，包项没有任何特定的次序。所以，在用作模拟的教室中，我们假定学生不按特定的次序来坐。如果你想逃课，且没有坐在链的第一把椅子上，那么，我们就不

一定非要删除你的椅子。相反，可做下列步骤：

- 1) 让坐在第一把椅子上的学生移到你之前坐的椅子上。
- 2) 使用情形 1 描述的步骤删除第一把椅子。

实际上，将情形 2 转化为情形 1，而对于后者我们是知道如何处理的。



学习问题 7 从有 5 把椅子的链中删除第一把椅子所需的步骤是什么？

学习问题 8 从有 5 把椅子的链中删除第三把椅子所需的步骤是什么？

方法 remove 和 clear

删除一个未指定的项。不带参数的 `remove` 方法从非空的包中删除一个未指定的项。根据第 1 章程序清单 1-1 的接口中给出的方法的规范说明，方法将返回它删除的项： 3.21

```
/** Removes one unspecified entry from this bag, if possible.
 * @return Either the removed object, if the removal was successful,
 *         or null. */
public T remove()
```

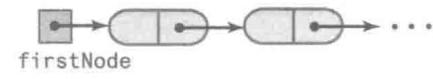
如果方法执行前包是空的，则方法返回 `null`。

从包中删除一个项，涉及从结点链中删除它。因为从链中容易删除第一个结点，所以可以将 `remove` 定义为让它删除第一个结点中的项。为此，采用下列步骤：

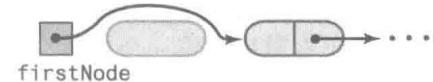
- 访问第一个结点的项，因此可以返回它。
- 设置 `firstNode` 指向第二个结点，如图 3-10 所示。如果不存在第二个结点，则 `firstNode` 设置为 `null`。
- `numberOfEntries` 减 1。

注意在下列 `remove` 的 Java 定义中是如何实现这些步骤的：

```
public T remove()
{
    T result = null;
    if (firstNode != null)
    {
        result = firstNode.data;
        firstNode = firstNode.next; // Remove first node from chain
        numberOfEntries--;
    } // end if
    return result;
} // end remove
```



a) 结点链



b) 删除了第一个结点后的结点链

图 3-10 删除第一个结点前后的结点链

我们首先比较 `firstNode` 是否等于 `null`，以此来检查链是否为空。说明一下，也可以调用 `isEmpty` 方法来进行检查。虽然访问首结点中的数据及将项数减 1 的 Java 语句，都是非常简单的语句，但语句

```
firstNode = firstNode.next;
```

的作用或许并不那么直观。如果链的第二个结点存在，这条语句让 `firstNode` 指向链的第二个结点，这应该很清楚。但如果不存在时会怎样呢？即当链中只含有一个结点时会怎样？这种情况下，`firstNode.next` 是 `null`，所以这个语句按要求将 `firstNode` 设置为 `null`。



注：再论方法 remove 的行为

作为第1章ADT包的设计者，我们不关心remove方法从包中删除的是哪一项。实际上，这个方法的规范说明是

如果可能，从包中删除一个未指定的对象

这个操作刚好删除了它能删除的任意项。本质上，让ADT的实现来选择到底删除哪个对象。如第2章及本章前面的内容所见，我们作为实现者，选择最容易删除的对象来实现。因为我们的选择没有特指，所以可以换为删除其他的项，比如一个随机的对象，或是最后添加的对象。

在设计ADT时，可以规范说明前面的这两种可能。即不是从包中删除任意的一个未指定对象，而是让ADT包具有下列行为：

从包中删除一个随机的对象

从包中删除最后添加的对象

最后一个行为像是“撤销”操作，它让包有了“记忆”。一般地，包没有这么复杂，但第5章的另一个ADT将有这个能力。

3.22 **删除给定的项。**如第1章程序清单1-1中接口所规范说明的，第二个remove方法删除给定的项并根据这个操作成功与否返回真或假：

```
/** Removes one occurrence of a given entry from this bag, if possible.  
 * @param anEntry The entry to be removed.  
 * @return True if the removal was successful, or false otherwise. */  
public boolean remove(T anEntry)
```

方法执行前如果包为空，或如果anEntry不在包中，则方法返回假。

要在结点链表中删除指定的项，首先必须找到这个项。即必须遍历链表，并检查结点中的项。假定我们在结点N中找到所要找的项。从前面段3.20关于教室的讨论中知道，如果结点N不在链表的第一个位置，则可用下面的步骤删除它的项：

- 1) 用第一个结点中的项替换结点N中的项。
- 2) 从链中删除第一个结点。

如果结点N是链表的第一个又会怎样呢？如果不单独处理这种情况，则前面的步骤中将用自己来替换第一个结点中的项。出现这样的情况，也比增加逻辑判断结点N是否是第一个结点要更简单些。

由此，得到方法remove的下列伪代码：

```
查找含有anEntry的结点N  
if (结点N存在)  
{  
    使用第一个结点中的项替换结点N中的项  
    从链表中删除首结点  
}  
根据操作是否成功返回真或假
```

3.23 **继续删除给定的项。**查找含有给定项的结点，与段3.17中contains方法中所做的事情一样。我们不在remove方法中重复这段代码，而是将它放到一个新的remove和contains都能调用的私有方法中。这个私有方法的定义如下：

```
// Locates a given entry within this bag.  
// Returns a reference to the node containing the entry, if located,  
// or null otherwise.
```

```

private Node getReferenceTo(T anEntry)
{
    boolean found = false;
    Node currentNode = firstNode;
    while (!found && (currentNode != null))
    {
        if (anEntry.equals(currentNode.data))
            found = true;
        else
            currentNode = currentNode.next;
    } // end while
    return currentNode;
} // end getReferenceTo

```

现在将前一段给出的 remove 方法的伪代码转换为如下的 Java 语句：

```

public boolean remove(T anEntry)
{
    boolean result = false;
    Node nodeN = getReferenceTo(anEntry);
    if (nodeN != null)
    {
        nodeN.data = firstNode.data; // Replace located entry with entry
                                     // in first node
        firstNode = firstNode.next; // Remove first node
        numberofEntries--;
        result = true;
    } // end if
    return result;
} // end remove

```



学习问题 9 不调用 getReferenceTo 方法，remove 方法能调用段 3.17 给出的 contains 方法原来的定义吗？解释之。

学习问题 10 修改方法 contains 的定义，让它调用私有方法 getReferenceTo。

学习问题 11 修改方法 getReferenceTo 的定义，让它使用计数器及 numberofEntries 而不是使用 currentNode 来控制循环。

学习问题 12 与上一个问题中描述的 getReferenceTo 方法相比，前一段中给出的方法定义的好处有哪些？

方法 clear。在前一章给出的类 ArrayBag 中，方法 clear 调用方法 remove 和 isEmpty 从包中删除所有的项。因为这个定义不依赖于包的表示方式，所以在 LinkedBag 中我们可以使用同样的定义。故 clear 的定义如下：

3.24

```

public void clear()
{
    while (!isEmpty())
        remove();
} // end clear

```



学习问题 13 方法 clear 的下列版本是不是释放了链表中的所有结点，故而得到了一个空包？解释之。

```

public void clear()
{
    firstNode = null;
} // end clear

```

**注：释放内存**

方法 `remove` 从链表中删除一个结点后，就没有引用再指向被删结点了，所以不能再使用它。另外，如附录 B 的段 B.20 所说明的，Java 运行时环境自动释放并回收分配给这样的结点的内存。程序员不需要，实际上也不可能写显式语句来释放空间。

**设计决策：LinkedBag 应该限制包的容量吗？**

虽然第 2 章提出的 ADT 包基于数组的实现方式中，禁止包的容量超出设置的限额，但 `LinkedBag` 中却并不这样处理。与 `ArrayBag` 不一样，`LinkedBag` 没有分配一个足够大的数组用作数据域，来保存预期的包项。而是这些项的链表按需每次增加一个结点。如果添加失败，则现有的链表保持不变。

虽然我们选择让 `LinkedBag` 对象有不受限制的容量，但如果你希望在有限的内存状态下使用 `LinkedBag`，仍可以限制 `LinkedBag` 的容量。

有设置和获取方法的类 Node

因为 `Node` 是类 `LinkedBag` 的内部类，所以 `LinkedBag` 可以直接按名访问 `Node` 的私有数据域。这样做使得写、读及理解实现的代码更加容易，特别是对新入门的 Java 程序员而言。但是，你真的应该通过调用访问方法和赋值方法（`set` 和 `get`）来访问类的数据域。事实上，在本书的其他部分我们都会这样做。本节给 `Node` 增加这几个方法，并探讨定义这个类的三种方式。

3.25

作为内部类。假定我们在程序清单 3-1 所示的内部类 `Node` 中增加 `getData`、`setData`、`getNextNode` 和 `setNextNode` 方法，则这个类如程序清单 3-4 所示。

程序清单 3-4 有设置和获取方法的内部类 `Node`

```

1  private class Node
2  {
3      private T data; // Entry in bag
4      private Node next; // Link to next node
5
6      private Node(T dataPortion)
7      {
8          this(dataPortion, null);
9      } // end constructor
10
11     private Node(T dataPortion, Node nextNode)
12     {
13         data = dataPortion;
14         next = nextNode;
15     } // end constructors
16
17     private T getData()
18     {
19         return data;
20     } // end getData
21
22     private void setData(T newData)
23     {
24         data = newData;
25     } // end setData
26
27     private Node getNextNode()
28     {

```

```

29         return next;
30     } // end getNextNode
31
32     private void setNextNode(Node nextNode)
33     {
34         next = nextNode;
35     } // end setNextNode
36 } // end Node

```

给 Node 添加了这些方法后，做如下的改变从而修改 LinkedBag 的实现：

3.26

- 将

```
newNode.next = firstNode;
```

改为

```
newNode.setNextNode(firstNode);
```

- 将

```
currentNode = currentNode.next;
```

改为

```
currentNode = currentNode.getNextNode();
```

- 将

```
result = firstNode.data;
```

改为

```
result = firstNode.getData();
```

- 将

```
entryNode.data = firstNode.data;
```

改为

```
entryNode.setData(firstNode.getData());
```

本章结尾的项目 2 要求你完成对 LinkedBag 的这些修改。

3.27

作为包内的一个类。像刚刚提到的这样修改 Node 和 LinkedBag 后，Node 仍然可以是一个私有内部类。因为 Node 是我们想要隐藏的实现细节，所以让它成为内部类是合适的。但如果我们改变想法，想在 LinkedBag 之外定义 Node，保留上一段中对 LinkedBag 所做的修改就可以了。我们可以——仅需很少的修改——让 Node 仅能在包内访问，或者甚至可以让它成为公有类。

将程序清单 3-4 中所给的 Node，转变为仅能被所在包内的其他类访问的类，首先要掉所有的访问修改符，用于数据域的访问修改符除外。然后将 $<T>$ 加在类定义中的每个 Node 之后，用作构造方法名的那个 Node 除外。修改后的类列在程序清单 3-5 中。

程序清单 3-5 具有包访问权限的类 Node

```

1 package BagPackage;
2 class Node<T>
3 {
4     private T      data;

```

```

5   private Node<T> next;
6
7   Node(T dataPortion) // The constructor's name is Node, not Node<T>
8   {
9     this(dataPortion, null);
10    } // end constructor
11
12  Node(T dataPortion, Node<T> nextNode)
13  {
14    data = dataPortion;
15    next = nextNode;
16  } // end constructor
17
18  T getData()
19  {
20    return data;
21  } // end getData
22
23  void setData(T newData)
24  {
25    data = newData;
26  } // end setData
27
28  Node<T> getNextNode()
29  {
30    return next;
31  } // end getNextNode
32
33  void setNextNode(Node<T> nextNode)
34  {
35    next = nextNode;
36  } // end setNextNode
37 } // end Node

```

3.28 如果 `LinkedBag` 类和 `Node` 类在同一个包中，且对 `LinkedBag` 类稍做修改，则类 `LinkedBag` 就可以访问程序清单 3-5 中所给的 `Node`。`Node` 在 `LinkedBag` 中的每次出现，现在都必须修改为 `Node<T>`。现在来修改 `LinkedBag`，在程序清单 3-6 中标注出这些修改。

程序清单 3-6 当 `Node` 在同一个包内时的类 `LinkedBag`

```

1 package BagPackage;
2 public class LinkedBag<T> implements BagInterface<T>
3 {
4   private Node<T> firstNode;
5   ...
6
7   public boolean add(T newEntry)           ← 这个地方出现的T是可选的，但尖括号
8   {                                     必须有。
9     Node<T> newNode = new Node<T>(newEntry);
10    newNode.setNextNode(firstNode);
11    firstNode = newNode;
12    numberEntries++;
13
14    return true;
15  } // end add
16  ...
17 } // end LinkedBag

```

本章结尾的项目 3 要求你完成这个版本的 `LinkedBag`。

3.29 作为具有声明了泛型的内部类。程序清单 3-6 中描述的 `LinkedBag` 的版本，可以将 `Node` 定义为一个内部类。`Node` 类似于程序清单 3-5 中所给的类，但需要做下列修改：

- 省去 package 语句。
- 让类、构造方法及方法都是私有的。
- 将泛型 T 替换为另一个标识符，例如 S。

因为 `LinkedBag` 和 `Node` 都声明了泛型，所以必须使用不同的标识符来表示它们。

本章结尾的项目 4 要求你按这里所描述的要求来修改 `Node` 和 `LinkedBag`。

使用链表实现 ADT 包的优缺点

你已经看到了如何使用链表来实现 ADT 包。这种方法最大的一个优点是链表可以按需来改变大小，所以包也是如此。只要内存可用，你可以在链表中添加想要的任意多的结点。另外，可以删除并回收不再需要的结点。尽管可以变长数组从而让包变大——如前一章所描述的那样——但每次都需要一个更大的数组，且必须将项从已满的数组拷贝到新数组中。使用链表时不需要这样的拷贝。3.30

将新项添加到数组尾或是链表头，都是相对简单的任务。这两个操作都很快，除非需要变长数组。类似地，从数组尾或链表头删除一项，也花费同样的时间。但是，删除指定项需要在数组或链表中进行查找。

最后，对于同样的长度，链表比数组需要更多的内存。虽然两种数据结构中都含有指向数据对象的引用，但在链表的每个结点中还含有指向另一个结点的引用。不过，数组常常比所需要的大，所以内存也是浪费的。链表仅按需使用内存。



学习问题 14 比较本章类 `LinkedBag` 中的 `contains` 方法和第 2 章类 `Resizable-ArrayBag` 中的 `contains` 方法。执行时一个比另一个花费更多的时间吗？解释之。

本章小结

- 使用称为结点的对象可以形成数据链表。每个结点有两部分。一部分含有指向数据对象的引用，第二部分指向链表中的下一结点。不过最后一个结点不指向其他结点而是含有 `null`。链表外部的头引用指向首结点。
- 修改两个引用就可以将一个结点添加到结点链表的开头：要添加的结点内的引用和链表头引用。
- 将链表的头引用设置为首结点内的引用，就可以删除结点链表的首结点。那个引用指向了链表中原来的第二个结点，使得第二个结点成为新的首结点。
- 找到结点链表中的某个结点，需要对链表进行遍历。从首结点开始，一个结点一个结点地顺序移动，直到找到所要的结点为止。
- 类 `Node` 可以是 `LinkedBag` 的内部类，或与 `LinkedBag` 在同一个包中。后一种情况下，`Node` 必须定义设置方法和获取方法，为它的数据域提供访问权限。

程序设计技巧

- 如果 `ref` 是指向链表中结点的引用，要保证在用 `ref` 访问 `ref.data` 或 `ref.next` 之前，它不能是 `null`。

练习

1. 为类 `LinkedBag` 添加一个构造方法，由给定的对象数组构造一个包。
 2. 考虑段 3.12 给出的 `LinkedBag` 中 `add` 方法的定义。交换方法体中第二条和第三条语句，如下：

```
firstNode = newNode;  
newNode.next = firstNode;
```

- a. 在修改后的 `LinkedBag` 的客户中，如下的语句将显示什么？

```
BagInterface<String> myBag = new LinkedBag<>();
myBag.add("30");
myBag.add("40");
myBag.add("50");
myBag.add("10");
myBag.add("60");
myBag.add("20");
int numberOfEntries = myBag.getCurrentSize();
Object[] entries = myBag.toArray();
for (int index = 0; index < numberOfEntries; index++)
    System.out.print(entries[index] + " ");
```

- b. 因为修改了方法 `add`, 如果有, `LinkedBag` 中的哪些方法在执行时会因这个修改而受到影响?
为什么?

对类 `LinkedBag`, 重做第 2 章的练习 2。

修改段 3.21 中给出的 `remove` 方法的定义, 让它从包中删除一个随机项。这个修改会影响到类 `LinkedBag` 中的其他方法吗?

为类 `LinkedBag` 定义方法 `removeEvery`, 从包中删除给定项的所有出现。

为类 `LinkedBag` 重做第 2 章中的练习 10。

为类 `LinkedBag` 重做第 2 章中的练习 11。

为类 `LinkedBag` 定义方法 `equals`。关于这个方法的细节参见第 2 章练习 12。

为类 `LinkedBag` 定义方法 `union`, 如第 1 章练习 5 所描述的。

0. 为类 `LinkedBag` 定义方法 `intersection`, 如第 1 章练习 6 所描述的。

1. 为类 `LinkedBag` 定义方法 `difference`, 如第 1 章练习 7 所描述的。

2. 在双向链表 (doubly linked chain) 中, 每个结点既可以指向前一个结点也可以指向后一个结点

图 3-11 显示一个双向链表及它的头引用。定义表示双向链表中的结点类。将这个类定义为实现 ADT 包的类的内部类。可以不设置方法和获取方法。

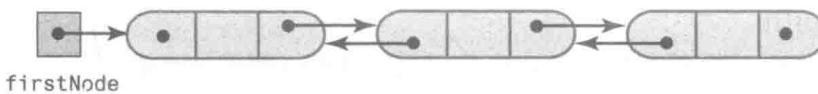


图 3-11 练习 12、练习 13、练习 14 和练习 15，及项目 7 中用到的双向链表

13. 重做练习 12，将类与实现 ADT 包的类定义在同一个包内。需要设置方法和获取方法。
 14. 列出将一个结点添加到图 3-11 所示的双向链表开头所需的步骤。
 15. 列出从图 3-11 所示的双向链表开头删除一个结点所需的步骤。

项目

1. 写程序，对类 `LinkedBag` 进行全面的测试。
 2. 程序清单 3-4 展示了带设置方法和获取方法的内部类 `Node`。修改类 `LinkedBag`，让它调用这些设置方法和获取方法，而不是按名直接访问私有数据域 `data` 和 `next`。
 3. 程序清单 3-5 展示 `Node` 是 `LinkedBag` 所在包内的类。使用 `Node` 的这个版本来修改

LinkedBag。

4. 修改段 3.29 描述的 Node 和 LinkedBag。
5. 定义表示集合的类 LinkedSet，实现第 1 章程序清单 1-5 中所给的接口。在你的实现中使用类 LinkedBag。然后写程序，充分展示你的实现。
6. 使用结点链表代替类 LinkedBag 重做前一个项目。
7. 定义类 DoublyLinkedListBag，使用图 3-11 所示的双向链表实现 ADT 包。使用练习 12 中定义的结点的内部类。
8. 重做前一个项目，但在结点的内部类内定义设置方法和获取方法。
9. 使用本章定义的或前面的项目中描述的集合或包的类，创建一个拼写检查器。细节参考第 2 章的项目 7 和项目 8。
10. 重做第 2 章的项目 4，使用结点链表替代数组。
11. 重做第 2 章的项目 5，使用结点链表替代数组。
12. 重做第 2 章的项目 6，使用结点链表替代数组。
13. (游戏) 你的公司正在开发一款游戏，其中的角色是 Sammie，依据角色找到或丢失珠宝的情况，每次 Sammie 的名字都会改变。例如当 Sammie 找到一个 opal 然后又找到一个 ruby，则它的名字先改为 SammieOpal 后改为 SammieOpalRuby。如果 SammieOpalRuby 丢失了 opal，则它的名字改为 SammieRuby。游戏中用到的珠宝有 opal、ruby、emerald、onyx、sapphire、jade 和 peridot。使用链表表示 Sammie 的名字，实现类 SammieName。类应该提供的方法有：在名字后添加珠宝、从名字中删除珠宝，及将当前的名字作为字符串返回。
14. (财务) 使用基于链式的实现方式替代基于数组的实现方式，重做第 2 章项目 10。
15. (财务) 日志是数据库中发生的事务的拷贝，用于数据库的备份。如果数据库毁坏了，可将日志中的事务重做一遍，以重构数据库。设计并实现类 AccountJournal，用来记录序言中项目 14a 所设计的事务。应该能向日志中添加事务、能重做事务。还应该能清除日志中的所有事务。
16. (电子商务) 使用 LinkedBag 替代 ArrayBag，重做第 2 章项目 11。

第 4 章 |

Data Structures and Abstractions with Java, Fifth Edition

算法的效率

先修章节：附录 B、第 2 章、第 3 章

目标

学习完本章后，应该能够

- 评估给定算法的效率
- 比较两个方法预期的执行时间，给出其算法的效率

制造商以惊人的速度推出比之前的产品更快、内存也更大的新计算机。可是我们——可能还有教你计算机的教授——要求你写出有效利用时间和空间（内存）的代码。这样的效率问题仍如早期计算机时代那样紧迫，那时的计算机比现在的慢得多，内存也少得多。当今，要求计算机去处理的海量数据集要求我们保持同样的效率准则。效率仍然是一个问题——某些情况下还是关键问题。

本章介绍计算机科学家用来衡量算法效率的术语及方法。有了这些背景，你不只能直观感受效率，还能定量谈论效率。

动机

4.1 示例。你可能认为，近期不会写一个执行时间特别长的程序。这或许是对的，但我们要给你看一些要花很长时间才能完成计算的简单的 Java 代码。

对任意正整数 n ，考虑求和问题 $1+2+\dots+n$ 。图 4-1 中含有 3 个求解这个问题的伪代码。算法 A 从左至右求和 $0+1+2+\dots+n$ 。算法 B 计算 $0+(1)+(1+1)+(1+1+1)+\dots+(1+1+\dots+1)$ 。最后，算法 C 使用代数恒等式来计算和。

| 算法 A | 算法 B | 算法 C |
|---|--|----------------------------------|
| <pre>sum = 0 for i = 1 to n sum = sum + i</pre> | <pre>sum = 0 for i = 1 to n { for j = 1 to i sum = sum + 1 }</pre> | <pre>sum = n * (n + 1) / 2</pre> |

图 4-1 对整数 $n > 0$ ，计算和 $1+2+\dots+n$ 的三个算法

4.2 现在将这些算法翻译为 Java 代码。如果我们使用 long 型整数，则可以写下列语句：

```
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // Ten thousand

// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
    sum = sum + i;
System.out.println(sum);

// Algorithm B
```

```

sum = 0;
for (long i = 1; i <= n; i++)
{
    for (long j = 1; j <= i; j++)
        sum = sum + 1;
} // end for
System.out.println(sum);

// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);

```

如果对 $n = 10000$ 执行这段代码，每个算法都会得到正确的答案 50005000。现在将 n 改为 100000，再次执行代码。你会再次得到正确答案，这次是 5000050000。但是，应该注意到算法 B 计算结果时的延迟。现在试着让 n 的值为 1000000。你会再次得到正确答案——500000500000——但不得不等更长的时间才能从算法 B 得到结果。等待的时间太长了以至于你可能会怀疑哪儿出了问题。如果还没有怀疑，那就再试更大的 n 值。

前面算法 B 的示例代码花了太长的时间运行，比另外两个算法长得多。如果这是你唯一执行的算法，你该怎么办呢？使用一台更快的计算机？那或许是一种解决办法，很显然你应该用一个不同的算法。



注：正如前一个示例所表示的，即使一个简单的程序也可能效率非常低。



注：如果算法的执行时间比实际的要长，那么试着重新规划，让它的时间效率更高。

算法效率的衡量

前一节应该让你确信了程序的效率问题。我们如何衡量效率，以便能比较问题的不同求解方法？在前一节中，我们用 3 种不同方法计算了前 n 个连续整数的和。然后发现，随着 n 的增大，有一个方法明显地慢于另外两个方法。但一般来讲，在选定一个方案之前先实现几种不同的想法，会需要做太多的工作，不切实际。此外，程序的执行时间在某种程度上与所使用的计算机和程序设计语言相关。故在实现算法之前来衡量算法的效率会更好。

例如，假定你想去市中心的商店。你的选择是步行、驾车、让朋友带你或乘公交。哪种方式最好？首先，什么是你概念中的最好？是省钱、省时间、节省朋友的时间还是环境？我们姑且认为，对于你来说最快的方式就是最好的选择。定义了这个标准后，如何来评估你的选择？你肯定不想为找出哪种方式最快而把所有 4 种选择都试过来。这就像为同一个任务写 4 个不同的程序，以便你能衡量哪个最快一样。相反，你应该调查每种选择的“代价”，考虑距离、你能驾车的速度、其他交通工具的流量、交通灯前停车的次数、天气，等等，即应该考虑对代价影响最大的因素。

在确定哪个算法最优时也要进行这样的考虑。而且我们必须定义最优指的是什么。算法有时间和空间需求，称为 **复杂度** (complexity)，这是我们可以说量的。当评估算法的复杂度时，我们不是衡量它有多么难懂或困难。而是衡量算法的**时间复杂度** (time complexity) —— 运行它花了多少时间 —— 或它的**空间复杂度** (space complexity) —— 运行它需要多少内存。一般地，我们独立分析这些需求。所以一个“最优”的算法可能是最快的或使用内存最少的。

4.3

4.4



注：什么是最优的？

通常一个问题的“最优”方案需要平衡不同的标准，例如时间、空间、通用性、编程难易，等等。

衡量算法复杂度的过程称为 **算法分析** (analysis of algorithm)。我们专注于算法的时间复杂度，因为通常它比空间复杂度更重要。你应该知道，算法的时间复杂度和它的空间复杂度之间常存在反比关系。如果修改一个算法，让它节省执行时间，通常会需要更多的空间。如果减少了算法的空间需求，很可能会需要更多的运行时间。但有时，你既能节省时间又能节省空间。

算法复杂度的衡量应该易于计算，肯定比实现算法要容易。应该用问题规模来表示衡量结果。**问题规模** (problem size) 是算法要处理的项数。例如，如果查找一个数据集合，则问题规模是集合中的项数。这种方式下，将算法的相对代价表示为问题规模的函数。一般地，我们对大问题感兴趣；即使算法的效率不高，小问题也可能只花很少的时间。

4.5

要知道，你不能计算算法所需的实际时间。毕竟，你还没有用 Java 实现算法，也没有选择计算机。相反，你找到了问题规模的一个函数，它就像是算法实际的时间需求。所以，当某些因素使时间需求增大时，函数值也因同样的原因而增大，反过来也一样。可以说，函数值与时间需求成正比 (directly proportional)。这样的一个函数称为 **增长率函数** (growth-rate function)，因为它衡量当问题规模增大时，算法的时间需求如何增大。因为它们衡量的是时间需求，故增长率函数有正值。比较两个算法的增长率函数，可以了解对于大规模问题，一个算法是否快于另一个算法。

4.6

示例。再次考虑对正整数 n 计算 $1+2+\dots+n$ 的问题。图 4-1 给出了 3 个算法——A、B 和 C——来完成这个计算。算法 A 从左至右求和 $0+1+2+\dots+n$ 。算法 B 计算 $0+(1)+(1+1)+(1+1+1)+\dots+(1+1+\dots+1)$ ，算法 C 使用代数恒等式来计算和。执行段 4.2 中的 Java 代码，发现算法 B 是最慢的。我们现在不需要实际运行代码，想要预测这种行为。

我们如何才能知道哪个算法是最慢的？哪个是最快的？通过考虑问题规模及所花的代价，来回答这些问题。整数 n 衡量的是问题规模：当 n 增大时，加和涉及更多的项。为衡量算法的代价或时间需求，必须找到一个合适的增长率函数。为此，可以着手统计算法所需的操作个数。

例如，图 4-1 中的算法 A 含有伪代码语句

```
for i = 1 to n
```

这条语句表示下列循环控制逻辑：

```
i = 1
while (i <= n)
{
    .
    .
    i = i + 1
}
```

这个逻辑需要一条对 i 的赋值语句、 i 和 n 之间进行的 $n+1$ 次比较、 n 个对 i 的加法及另外 n 个对 i 的赋值。总起来，循环控制逻辑需要 $n+1$ 个赋值语句、 $n+1$ 次比较及 n 个加法。

此外，算法 A 的初始化和循环体还需要另外的 $n+1$ 个赋值语句和 n 个加法。加在一起，算法 A 需要 $2n+2$ 个赋值语句、 $2n$ 个加法和 $n+1$ 次比较。

这些不同的操作可能花费不同的执行时间。例如，如果每个赋值语句花费不多于 t_a 个单位时间，每个加法花费不多于 t_+ 个单位时间，而每次比较花费不多于 t_c 个单位时间，则算法 A 所需的时间不多于

$$(2n + 2)t_a + (2n)t_+ + (n + 1)t_c \text{ 个单位时间}$$

如果我们将 t_a 、 t_+ 和 t_c 都替换为 3 个值中最大的一个，且称它为 t ，则算法 A 需要的时间不多于 $(5n + 3)t$ 个单位时间。我们得出结论，算法 A 需要的时间与 $5n + 3$ 成正比。

但是，最重要的不是操作个数的精确计数，而是算法的一般行为。函数 $5n + 3$ 与 n 成正比。如你将要看到的，我们不需要计算每个操作，就能知道算法 A 需要随 n 线性增加的时间。

统计基本操作

算法的基本操作（basic operation）是那些对其总的时间需求贡献最大的因素。例如，图 4.7 中的算法 A 和 B 中，加法是其基本操作。查看数组中是否含有某个对象的算法中，比较是其基本操作。要知道，最频繁的操作不一定是基本操作。例如，赋值语句常常是算法中最频繁的操作，但它们很少是基本操作。

忽略非基本的操作，如变量初始化、控制循环的操作等，不会影响算法速度的最终结论。例如，算法 A 在循环体中需要 n 个将 i 加到 sum 的加法。虽然忽略了算法中那些非基本操作，仍可以得出结论，算法 A 需要的时间随 n 线性增长。

不论是看基本操作数 n ，还是总的操作数 $5n+3$ ，都能得出相同的结论：算法 A 需要的时间与 n 成正比。所以算法 A 的增长率函数是 n 。

 **继续看示例。**现在来统计算法 B 和 C 中需要的基本操作的个数。算法 B 中的基本操作是加法；算法 C 中，基本操作是加法、乘法和除法。图 4.8 中用表格列出了算法 A、B 和 C 所需的基本操作个数。记住，这些统计结果不包括控制循环的赋值和操作。前面的讨论应该能让你明白，我们可以忽略这些操作。

4.7

4.8

| | 算法 A | 算法 B | 算法 C |
|--------|------|-----------------|------|
| 加法 | n | $n(n + 1) / 2$ | 1 |
| 乘法 | | | 1 |
| 除法 | | | 1 |
| 总的基本操作 | n | $(n^2 + n) / 2$ | 3 |

图 4.2 图 4.1 中各算法所需基本操作的个数

算法 B 所需的时间与 $(n^2+n)/2$ 成正比，算法 C 需要的时间是不依赖于 n 值的一个常数。时间需求作为 n 的函数画在图 4.3 中。从这个图可以看出，随着 n 的增大，算法 B 需要的时间最多。

 **学习问题 1** 对于任意的正整数 n ，分析算法时会遇到的一个恒等式是
 $1 + 2 + \dots + n = n(n + 1)/2$

你能推导出它吗？如果可以，就不需要记住它。提示：写下 $1+2+\dots+n$ 。在它的下面写下 $n+(n-1)+\dots+1$ 。然后自左至右相加。

学习问题 2 你能推导出图 4-2 中的值吗？提示：对于算法 B，使用学习问题 1 中给出的恒等式。

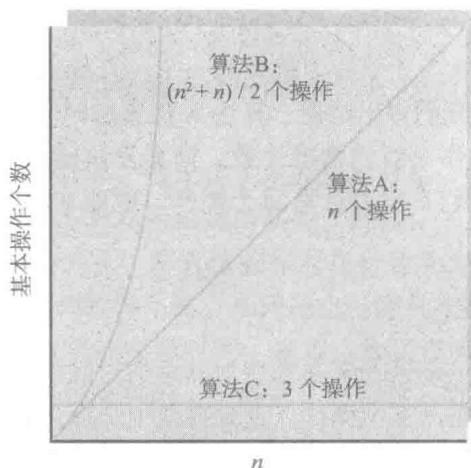


图 4-3 图 4-1 中各算法所需的基本操作个数表示为 n 的函数

注：有用的恒等式

$$1+2+\dots+n=n(n+1)/2$$

$$1+2+\dots+(n-1)=n(n-1)/2$$

4.9

典型的增长率函数都是简单的代数公式。为什么？回忆一下，因为当问题很小时，效率低下的算法的影响不会引起你的注意，故你关心的都应该是大问题。所以，当对算法进行比较时，如果我们只关心大的 n 值，则可以只考虑每个增长率函数中的主项。

例如，当 n 变大时， $(n^2 + n)/2$ 的表现与 n^2 一样。首先，对于大的 n 值， n^2 比 n 大得多，所以 $(n^2 + n)/2$ 的表现与 $n^2/2$ 一样。其次，当 n 变大时， $n^2/2$ 的表现与 n^2 一样。换句话说，对于大的 n ， $(n^2 + n)/2$ 与 n^2 的值差相对不大，且可被忽略。故不使用 $(n^2 + n)/2$ 而是使用 n^2 ——取最大指数——作为算法 B 的增长率函数，且称算法 B 需要的时间与 n^2 成比例。另一方面，算法 C 需要的时间不依赖于 n ，之前我们已知道，算法 A 需要的时间与 n 成比例。得到结论，算法 C 是最快的，而算法 B 是最慢的。

注：常见增长率函数的相对大小

当 $n > 10$ 时，可能遇到的增长率函数的增长量级如下：

$$1 < \log(\log n) < \log n < \log^2 n < n < n \log n < n^2 < n^3 < 2^n < n!$$

这里给出的对数的底是 2。在后面段 4.16 中将会看到，底的选择不是问题。

图 4-4 的表格中列出了当问题规模 n 增大时这些函数的量级。从这些数据可以看到，其增长率函数是 $\log(\log n)$ 、 $\log n$ 或 $\log^2 n$ 的算法，比起其增长率函数是 n 的算法花少得多的时间。不过 $n \log n$ 的值明显大于 n ，而这两个函数描述的增长率明显快于 n^2 。

| n | $\log(\log n)$ | $\log n$ | $\log^2 n$ | n | $n \log n$ | n^2 | n^3 | 2^n | $n!$ |
|--------|----------------|----------|------------|-----------|------------|-----------|-----------|----------------|------------------|
| 10 | 2 | 3 | 11 | 10 | 33 | 10^2 | 10^3 | 10^3 | 10^5 |
| 10^2 | 3 | 7 | 44 | 100 | 664 | 10^4 | 10^6 | 10^{30} | 10^{94} |
| 10^3 | 3 | 10 | 99 | 1000 | 9966 | 10^6 | 10^9 | 10^{301} | 10^{1435} |
| 10^4 | 4 | 13 | 177 | 10,000 | 132,877 | 10^8 | 10^{12} | 10^{3010} | $10^{19,335}$ |
| 10^5 | 4 | 17 | 276 | 100,000 | 1,660,964 | 10^{10} | 10^{15} | $10^{30,103}$ | $10^{243,338}$ |
| 10^6 | 4 | 20 | 397 | 1,000,000 | 19,931,569 | 10^{12} | 10^{18} | $10^{301,030}$ | $10^{2,933,369}$ |

图 4-4 在 n 值增大时典型的增长率函数的估值

注：当分析一个算法的时间效率时，考虑大的问题。对于小问题，同一问题的两个方案在执行时间上的差异通常微不足道。

最优、最差和平均情况

对于操作一个数据集的某些算法，执行时间依赖于数据集的大小。例如，在整数数组中查找最小整数的时间需求依赖于整数的个数，而不是整数本身。查找 100 个整数中的最小值，不管这些整数值是多少，所花的时间是一样的。

不过另一个算法的时间需求不依赖于数据集的大小，而是依赖于数据本身。例如，假定数组中含有某个值，我们想知道它在数组中的位置。假定查找算法检查数组中的每个值，直到找到要找的项。如果算法在它所检查的第一个元素找到这个值，则它只进行了一次比较。在这种最优情况（best case）下，算法所花的时间最少。算法做得不会比最优情况时间更好了。如果最优情况时间仍很慢，那么你需要另外一个算法。

现在假定算法在比较了数组中的每个值后找到了所需的值。这是算法的最差情况（worst case），因为它需要最多的时间。如果你可以忍受这个最差情况，则你的算法就是可接受的。对于许多算法，最差和最优情况很少出现。所以当它处理一个典型的数据集时，我们考虑算法的平均情况（average case）。算法的平均情况时间需求更有用，但很难估计。注意到，平均情况时间不是最优情况时间和最差情况时间的平均值。

注：有些算法的时间需求依赖于所给的数据值。那些时间介于最小或最优情况时间到最大或最差情况时间之间。一般地，最优和最差情况不会出现。对于这种算法的时间需求，更有用的衡量是它的平均情况时间。

但是，有些算法没有最优、最差及平均情况。它们的时间需求依赖于所给数据项的个数，而不是数据值。

大 O 表示

计算机科学家使用一种符号表示算法的复杂度。例如，考虑图 4-1 所给的算法 A、B 和 C，及图 4-2 中显示的每个算法需要的基本操作个数。不是说算法 A 有与 n 成比例的时间需求，而是说 A 是 $O(n)$ 的。我们称这个符号为大 O，因为它使用大写的字母 O。我们将 $O(n)$ 读作“ n 的大 O”或“最多 n 阶”。类似地，因为算法 B 有与 n^2 成正比的时间需求，我们说 B 是 $O(n^2)$ 的。算法 C 总是需要 3 个基本操作。不管问题规模 n 如何，这个算法需要相同的时间。我们说算法 C 是 $O(1)$ 的。

4.12



示例。设想你在婚礼现场，坐在有 n 个人的一张桌边。为了祝酒，一位侍者把香槟倒进每一个杯子里。这个工作是 $O(n)$ 的。有人在致辞。哪怕致辞可能要用很长时间，但这是 $O(1)$ 的，因为它不依赖于客人的人数。如果与同桌的每个人碰杯，则执行的是一个 $O(n)$ 操作。如果同桌的每个人都这样做，则总共要执行 $O(n^2)$ 次碰杯。

4.13

大 O 符号有形式化的数学含义，可用来证明我们在前一节的讨论。你明白算法的实际时间需求与问题规模 n 的函数 f 成正比。例如， $f(n)$ 可能是 n^2+n+1 。这种情况下，可以说算法最多是 n^2 阶的，即 $O(n^2)$ 。基本上我们可以用一个更简单的函数——称它为 $g(n)$ ——来替换 $f(n)$ 。本例中， $g(n)$ 是 n^2 。

函数 $f(n)$ 具有最多 $g(n)$ 阶，即 $f(n)$ 是 $O(g(n))$ 或 $f(n)=O(g(n))$ ——的真正含义是什么？形式上，它的含义由下列数学定义描述：



注：大 O 的形式化定义

函数 $f(n)$ 具有最多 $g(n)$ 阶——即 $f(n)$ 是 $O(g(n))$ ——如果

- 存在正实数 c 和正整数 N ，对于所有的 $n \geq N$ ，有 $f(n) \leq c \times g(n)$ 。即当 n 足够大时， $c \times g(n)$ 是 $f(n)$ 的上界。

简单来说， $f(n)$ 是 $O(g(n))$ 的，这意味着，当 n 充分大时， $c \times g(n)$ 提供了 $f(n)$ 的增长率的一个上界（upper bound）。对于足够大的所有数据集，算法总是需要少于 $c \times g(n)$ 个基本操作。

图 4-5 图示了大 O 形式定义中的函数值。可以看到，当 n 足够大时——即当 $n \geq N$ 时—— $f(n)$ 不会超出 $c \times g(n)$ 。对于较小的 n 值，可能是相反的结论。这不重要，因为我们可以忽略 n 的这些值。

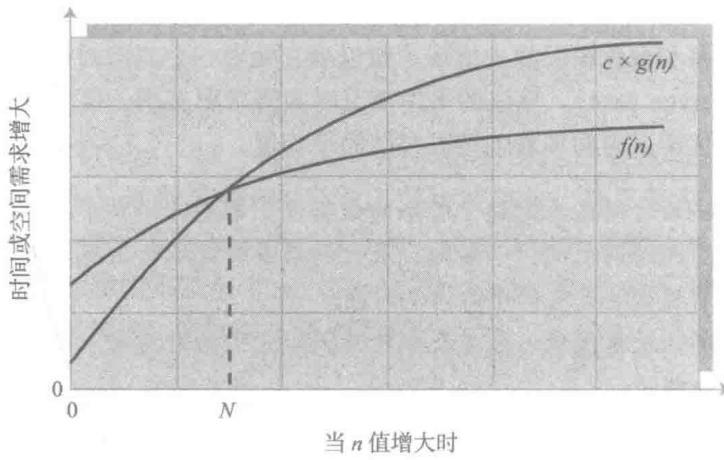


图 4-5 两个增长率函数值图

4.14



示例。在段 4.6 中我们提到过，如果算法使用 $5n+3$ 个操作，则它需要的时间与 n 成比例。现在可以使用大 O 的形式化定义来说明 $5n+3$ 是 $O(n)$ 的。

当 $n \geq 3$ 时， $5n+3 \leq 5n+n = 6n$ 。所以如果令 $f(n) = 5n+3$ ， $g(n) = n$ ， $c = 6$ 且 $N = 3$ ，则对 $n \geq 3$ ，有 $f(n) \leq 6g(n)$ ，或 $5n+3 = O(n)$ 。即如果算法需要的时间与 $5n+3$ 成正比，则它是 $O(n)$ 的。

常数 c 和 N 的其他值也是正确的。例如，当 $n \geq 1$ 时， $5n+3 \leq 5n+3n = 8n$ 。所以，选择 $c = 8$ 及 $N = 1$ ，可表明 $5n+3$ 是 $O(n)$ 的。

选择 $g(n)$ 时需要特别小心。例如，当 $n \geq 1$ 时，刚刚推出 $5n+3 \leq 8n$ 。但当 $n \geq 9$ 时，有 $8n < n^2$ 。所以为什么不让 $g(n) = n^2$ 且说算法是 $O(n^2)$ 的呢？虽然这个结论是正确的，但它没有达到能达到的好——或严格 (tight)。要得到 $f(n)$ 尽可能小的上界。

 注：算法时间需求的上界应该尽可能小，应该使用图 4-4 中所给的那样的简单函数。

 示例。我们来说明 $4n^2+50n-10$ 是 $O(n^2)$ 的。容易看到

4.15

对于任意的 n ， $4n^2+50n-10 \leq 4n^2+50n$ 。

因为对于 $n \geq 50$ ，有 $50n \leq 50n^2$ ，

故对于 $n \geq 50$ ，有 $4n^2+50n-10 \leq 4n^2+50n^2 = 54n^2$ 。

所以，取 $c = 54$ 及 $N = 50$ ，已说明 $4n^2+50n-10$ 是 $O(n^2)$ 的。

 注：要说明 $f(n)$ 是 $O(g(n))$ 的，将 $f(n)$ 中较小的项替换为更大的项，直到只剩下一个项时为止。

 学习问题 3 说明 $3n^2+2^n$ 是 $O(2^n)$ 的。你用的 c 和 N 的值分别是多少？

 示例：说明 $\log_b n$ 是 $O(\log_2 n)$ 的。让 $L = \log_b n$ 且 $B = \log_2 b$ 。根据对数的含义，有 $n = b^L$ 且 $b = 2^B$ 。综上，有

$$n = b^L = (2^B)^L = 2^{BL}$$

故对于任意的 $n \geq 1$ ，有 $\log_2 n = BL = B \log_b n$ ，或者 $\log_b n = (1/B) \log_2 n$ 。在大 O 的定义中，取 $c = 1/B$ 且 $N = 1$ ，结论得证。

虽然这个结论是从本例得出的，但对于一般的对数函数，不管基数是什么，结论都一样。通常，增长率函数中用到的对数的底都是 2。但因为底通常没有关系，故一般地我们省略它。

 注：增长率函数中对数的底常常省略，因为 $O(\log_a n)$ 是 $O(\log_b n)$ 的。

 注：恒等式

大 O 符号的下列恒等式成立：

$$O(kg(n)) = O(g(n)) \quad \text{对常数 } k$$

$$O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$$

$$O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$$

$$O(g_1(n) + g_2(n) + \dots + g_m(n)) = O(\max(g_1(n), g_2(n), \dots, g_m(n)))$$

$$O(\max(g_1(n), g_2(n), \dots, g_m(n))) = \max(O(g_1(n)), O(g_2(n)), \dots, O(g_m(n)))$$

使用这些恒等式并忽略增长率函数中较小的项，通常稍做努力便可找到算法时间需求的阶。例如，如果增长率函数是 $4n^2 + 50n - 10$ ，则

$$O(4n^2 + 50n - 10) = O(4n^2) \quad \text{忽略较小的项}$$

$$= O(n^2) \quad \text{忽略常数系数}$$



学习问题 4 对于 $k > 0$ 且 $n > 0$, 如果 $P_k(n) = a_0n^k + a_1n^{k-1} + \dots + a_k$, 则 $O(P_k(n))$ 是多少?

程序结构的复杂度

4.17 算法或程序中语句序列的时间复杂度是语句各自复杂度的和。不过找到这些复杂度中的最大的就足够了。一般地, 如果 S_1, S_2, \dots, S_k 是程序语句序列, 且如果 g_i 是语句 S_i 的增长率函数, 则序列的时间复杂度是 $O(\max(g_1, g_2, \dots, g_k))$, 这等于 $\max(O(g_1), O(g_2), \dots, O(g_k))$ 。

if 语句

```
if (条件)
    S1
else
    S2
```

的时间复杂度是条件 (*condition*) 的复杂度加上 S_1 或 S_2 中复杂度最大者。

循环的时间复杂度是循环体的复杂度乘上循环体的执行次数。所以如下循环

```
for i = 1 to m; i = i + 1
    S
```

的复杂度是 $O(m \times g(n))$ 或 $m \times O(g(n))$, 其中 $g(n)$ 是 S 的增长率函数。注意, 本例中循环变量 i 每次加 1。下列循环中, i 每次加倍

```
for i = 1 to m; i = 2 * i
    S
```

这个循环的复杂度是 $O(\log(m) \times g(n))$ 或 $O(\log(m)) \times O(g(n))$ 。



注: 程序结构的复杂度

| 结构 | 时间复杂度 |
|--|---------------------------------------|
| 顺序的程序语句 S_1, S_2, \dots, S_k , 其增长率函数分别是 g_1, \dots, g_k | $\max(O(g_1), O(g_2), \dots, O(g_k))$ |
| 在增长率函数分别是 g_1 和 g_2 的程序语句 S_1 和 S_2 中进行选择的 if 语句 | $O(\text{条件}) + \max(O(g_1), O(g_2))$ |
| 迭代 m 次且循环体的增长率函数是 g 的循环 | $m \times O(g(n))$ |



注: 其他记号

虽然本书中常使用大 O 记号, 但在描述算法的时间需求 $f(n)$ 时, 有时也使用其他记号。这里向你简单地介绍一下。从之前见过的大 O 的定义开始, 然后还定义大 Ω 和大 Θ 。

- **大 O 。** $f(n)$ 具有最多 $g(n)$ 的阶——即 $f(n)$ 是 $O(g(n))$ 的——如果存在正常数 c 和 N , 对于所有的 $n \geq N$, 有 $f(n) \leq c \times g(n)$ 。即 $c \times g(n)$ 是时间需求 $f(n)$ 的上界。换句话说, $f(n)$ 不大于 $c \times g(n)$ 。所以使用大 O 的分析得到算法最大的时间需求。
- **大 Ω 。** $f(n)$ 具有至少 $g(n)$ 的阶——即 $f(n)$ 是 $\Omega(g(n))$ 的——如果 $g(n)$ 是 $O(f(n))$ 的。换句话说, 如果存在正常数 c 和 N , 对于所有的 $n \geq N$, 有 $f(n) \geq c \times g(n)$, 则 $f(n)$ 是 $\Omega(g(n))$ 的。时间需求 $f(n)$ 不小于其下界 $c \times g(n)$ 。所以使用大 Ω 的分析得到算法最小的时间需求。

- 大 Θ 。 $f(n)$ 具有 $g(n)$ 阶——即 $f(n)$ 是 $\Theta(g(n))$ 的——如果 $f(n)$ 是 $O(g(n))$ 且 $g(n)$ 是 $O(f(n))$ 的。换句话说，我们可以说， $f(n)$ 是 $O(g(n))$ 且 $f(n)$ 是 $\Omega(g(n))$ 的。时间需求 $f(n)$ 与 $g(n)$ 相同。即 $c \times g(n)$ 是 $f(n)$ 的下界也是上界。大 Θ 分析是我们能够得到的最好的时间需求分析。即便如此，大 O 是更常用的符号。

图示化效率

算法的大部分工作发生在重复阶段，即执行循环或是——如在第 9 章所见的——作为递归调用的结果。本节，将说明几个例子的时间效率。4.18

从图 4-1 中算法 A 的循环开始，它的伪代码如下：

```
for i = 1 to n
    sum = sum + i
```

循环体需要常量的执行时间，所以它是 $O(1)$ 的。图 4-6 使用一个图标表示了这个时间，所以一行内的 n 个图标表示循环的总执行时间。这个算法是 $O(n)$ 的：它的时间需求随着 n 的增大而增大。

图 4-1 中的算法 B 含有嵌套的循环，如下所示。

```
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

4.19

```
for i = 1 to n
    sum = sum + i
```

The diagram shows a sequence of n workers, each represented by a small icon of a person digging a hole. The workers are arranged horizontally, with labels 1, 2, 3, ..., n below them. This visual representation corresponds to the pseudocode for a loop that iterates n times, where each iteration involves a constant-time operation (digging a hole).

图 4-6 一个 $O(n)$ 算法

当循环嵌套时，先评估最内层的循环。这里，内层循环的循环体需要常量的执行时间，所以它是 $O(1)$ 的。如果还是用一个图标来表示这个时间，则一行内的 i 个图标表示内层循环的时间需求。因为内层循环是外层循环的循环体，故它执行 n 次。图 4-7 说明了这些嵌套循环的时间需求，它与 $1 + 2 + \dots + n$ 成比例。学习问题 1 要求你证明

$$1 + 2 + \dots + n = n(n+1)/2$$

这是 $n^2 / 2 + n / 2$ 。所以计算是 $O(n^2)$ 的。

```
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

The diagram illustrates the execution of nested loops. It shows a sequence of workers digging holes, grouped by the value of i . For $i=1$, there is one worker. For $i=2$, there are two workers. For $i=3$, there are three workers. This pattern continues up to $i=n$, where there are n workers. The workers are arranged in a grid-like structure, with labels 1, 2, 3, ..., n below them, representing the total number of iterations of the inner loop for each iteration of the outer loop.

图 4-7 一个 $O(n^2)$ 算法

4.20 前一段中内层循环的循环体依据外层循环执行不同的次数。假定改变内层循环，对外层循环的每次重复，内层都执行相同的次数，如下所示。

```
for i = 1 to n
{
    for j = 1 to n
        sum = sum + 1
}
```

图 4-8 说明了这些嵌套的循环，表明计算是 $O(n^2)$ 的。

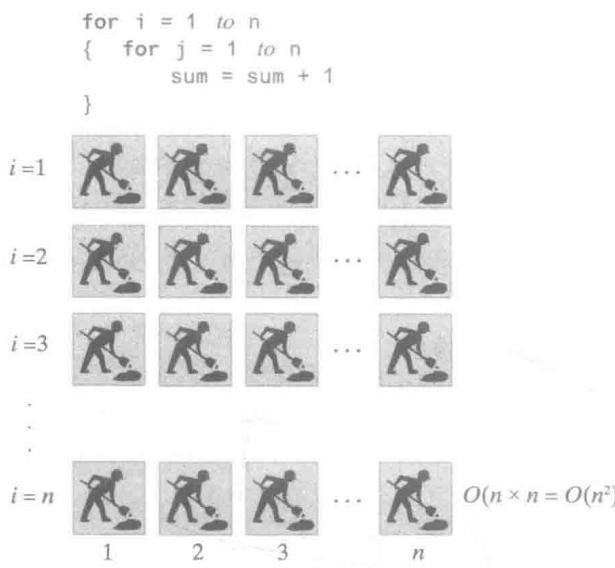


图 4-8 另一个 $O(n^2)$ 算法

学习问题 5 使用大 O 符号，下列计算的时间需求是多少阶？

 STUDY

```
for i = 1 to n
{
    for j = 1 to 5
        sum = sum + 1
}
```

4.21 让我们先感受一下图 4-4 中的增长率函数。正如我们提到的， $O(1)$ 算法的时间需求不依赖于问题规模 n 。可以将这样的算法应用于越来越大的问题而不影响执行时间。这种情形是理想的，但不是典型的。

对于其他的阶，如果问题规模增大一倍会怎样呢？对于 $O(\log n)$ 的算法，其时间需求会改变，但并不是很大。 $O(n)$ 算法需要两倍的时间， $O(n^2)$ 算法需要 4 倍的时间，而 $O(n^3)$ 算法需要 8 倍的时间。 $O(2^n)$ 算法的问题规模增大一倍，会需要平方的时间需求。这些结果列表在图 4-9 中。

学习问题 6 假定对某个确定规模的给定问题，可用 $O(n)$ 算法在时间 t 内求解。如果问题规模增大一倍，同样时间内求解这个问题，计算机需要快多少？

学习问题 7 使用 $O(n^2)$ 算法重做前一个学习问题。

| 大小为 n 的问题的增长率函数 | 大小为 $2n$ 的问题的增长率函数 | 时间需求的影响 |
|-------------------|--------------------|--------------|
| 1 | 1 | 无 |
| $\log n$ | $1 + \log n$ | 可以忽略不计 |
| n | 2 | 增大一倍 |
| $n \log n$ | $2n \log n + 2n$ | 增大一倍再加上 $2n$ |
| n^2 | $(2n)^2$ | 4 倍 |
| n^3 | $(2n)^3$ | 乘上 8 |
| 2^n | 2^{2n} | 平方 |

图 4-9 问题规模增大一倍时对算法时间需求的影响

现在假定，你的计算机每秒可执行 100 万次操作。采用一个算法求解规模为 100 万的一个问题要花多长时间？在不知道算法的情况下不能精确回答这个问题，但图 4-10 中的计算可让你知道，大 O 算法对结果的影响有多大。 $O(\log n)$ 的算法会花比 1 秒还少得多的时间，而 $O(2^n)$ 的算法将花好几万亿年！注意到，这些计算将 $O(g(n))$ 的时间需求估算为 $g(n)$ 。虽然这个近似不是普遍有效的，但对许多算法它还是合理的。

注：只要问题规模小，就可以使用 $O(n^2)$ 、
 $O(n^3)$ 甚至 $O(2^n)$ 的算法。例如，在速度为每秒 100 万次操作的机器上， $O(n^2)$ 的算法将花费 1 秒钟解决一个规模为 1000 的问题。 $O(n^3)$ 的算法将花费 1 秒钟解决一个规模为 100 的问题。而 $O(2^n)$ 的算法将花费大约 1 秒钟解决一个规模为 20 的问题。

学习问题 8 下列算法找出数组前 n 个元素中是否含有重复的项。最差情况下这个算法的大 O 表示是什么？

```
Algorithm hasDuplicates(array, n)
for index = 0 to n - 2
    for rest = index + 1 to n - 1
        if (array[index] equals array[rest])
            return true
return false
```

实现 ADT 包的效率

现在考虑前面章节讨论的 ADT 包两种实现的时间效率。

基于数组的实现

第 2 章给出的 ADT 包——ArrayBag——的一种实现是使用定长数组来表示包项。现在可以评估这种实现方式下包操作的效率。

向包中添加一项。从将新项添加到包中的操作开始。第 2 章段 2.15 提供了这个操作的下列实现：

| 增长率函数 g | $g(10^6) / 10^6$ | 4.22 |
|------------|-------------------|------|
| $\log n$ | 0.000 019 9 秒 | |
| n | 1 秒 | |
| $n \log n$ | 19.9 秒 | |
| n^2 | 11.6 天 | |
| n^3 | 31 709.8 年 | |
| 2^n | $10^{301\,016}$ 年 | |

图 4-10 采用不同阶的算法，在每秒处理 100 万次操作的机器上处理 100 万项的时间需求

```

public boolean add(T newEntry)
{
    checkIntegrity();
    boolean result = true;
    if (isArrayFull())
    {
        result = false;
    }
    else
    { // Assertion: result is true here
        bag[numberOfEntries] = newEntry;
        numberOfEntries++;
    } // end if
    return result;
} // end add

```

这个方法中的每一步——检查初始化是否完成、检测包是否满、将新项赋给数组元素及长度增1——都是 $O(1)$ 操作。那么这个方法是 $O(1)$ 的。直观上看，因为方法将新项正好添加在数组最后一项的后面，而我们知道含有新项的数组元素的下标。所以，可以进行这个赋值，而不影响包中的其他项。

4.24 在包中查找给定的项。ADT包有方法**contains**，它检测包中是否含有给定的项。第2章段2.32中给出的这个方法基于数组的实现如下：

```

public boolean contains(T anEntry)
{
    checkIntegrity();
    return getIndexOf(anEntry) > -1;
} // end contains

```

checkIntegrity的执行时间不依赖于包中的项数，所以是 $O(1)$ 的。要找的项如果存在，通过调用私有方法**getIndexOf**，方法**contains**找到数组中第一个含有这个项的元素。我们来检查第2章段2.31中描述的**getIndexOf**：

```

private int getIndexOf(T anEntry)
{
    int where = -1;
    boolean found = false;
    int index = 0;
    while (!found && (index < numberOfEntries))
    {
        if (anEntry.equals(bag[index]))
        {
            found = true;
            where = index;
        } // end if
        index++;
    } // end while
    return where;
} // end getIndexOf

```

这个方法在数组**bag**中查找给定的项**anEntry**。这个方法的基本操作是比较。如之前在段4.10中所描述的，假定包中含有 n 个项，最优情况下方法进行一次比较，最差情况下进行 n 次比较。一般地，方法将进行大约 $n/2$ 次比较。可以推断，方法**contains**最优情况下是 $O(1)$ 的，最差和平均情况都是 $O(n)$ 的。



注：为简化示例，我们考虑的是定长数组。一般地，基于数组的包可以按需改变数组的大小。数组扩大一倍是 $O(n)$ 操作。如第2章段2.38所说明的，接下来的 n 次添加将共同分摊倍增操作的开销。



学习问题 9 ArrayBag 的 remove 方法的大 O 是多少？假定用定长数组表示包，使用的一个参数类似于刚用在 contains 中的参数。

学习问题 10 重做学习问题 9，分析方法 getFrequencyOf。

学习问题 11 重做学习问题 9，分析方法 toArray。

链式实现

向包中添加一个项。现在考虑第 3 章给出的 ADT 包——LinkedBag——的链式实现。4.25 从段 3.12 中将项添加到包中的方法 add 开始：

```
public boolean add(T newEntry) // OutOfMemoryError possible
{
    // Add to beginning of chain:
    Node newNode = new Node(newEntry);
    newNode.next = firstNode;    // Make new node reference rest of chain
                                // (firstNode is null if chain is empty)
    firstNode = newNode;        // New node is at beginning of chain
    numberofEntries++;
    return true;
} // end add
```

这个方法内的所有语句都是 $O(1)$ 操作，所以方法是 $O(1)$ 的。

在包中查找给定项。第 3 章段 3.17 中所给的方法 contains，在结点链中查找给定项：4.26

```
public boolean contains(T anEntry)
{
    boolean found = false;
    Node currentNode = firstNode;
    while (!found && (currentNode != null))
    {
        if (anEntry.equals(currentNode.data))
            found = true;
        else
            currentNode = currentNode.next;
    } // end while
    return found;
} // end contains
```

当要找的项出现在结点链表的第一个结点中时出现最优情况。因为方法有指向链表首结点的引用，所以不需要遍历。故这种情况下方法是 $O(1)$ 的。

最差情况下，要遍历到链表的最后一个结点。这种情况下操作是 $O(n)$ 的。最后，一般情况或平均情况下，遍历将检查 $n/2$ 个结点，那么它是 $O(n)$ 操作。



注：查找链式实现的包

查找一个出现在结点链表开头的项是 $O(1)$ 操作。对链表的任意查找中，这种情况花的时间最少，故这种情况是最优情况。如果项在链表的最后一个结点中，那么查找它是 $O(n)$ 的。这个查找花的时间最多，所以这是最差情况。在结点链表中找到项所需的实际时间取决于哪个结点含有这个项。



学习问题 12 当查找一个不在包中的项时，LinkedBag 的方法 contains 的大 O 是多少？

学习问题 13 LinkedBag 的 `remove` 方法的大 O 是多少？使用一个类似于刚用在 `contains` 中的参数。

学习问题 14 重做学习问题 13，分析方法 `getFrequencyOf`。

学习问题 15 重做学习问题 13，分析方法 `toArray`。

两种实现的比较

4.27 使用大 O 符号，图 4-11 总结了分别使用定长数组和结点链表实现 ADT 包操作的时间复杂度。对某些操作，多个时间需求表示的是最优、最差和平均情况。

| 操作 | 定长数组 | 链式 |
|--|--------------------|--------------------|
| <code>add(newEntry)</code> | $O(1)$ | $O(1)$ |
| <code>remove()</code> | $O(1)$ | $O(1)$ |
| <code>remove(anEntry)</code> | $O(1), O(n), O(n)$ | $O(1), O(n), O(n)$ |
| <code>clear()</code> | $O(n)$ | $O(n)$ |
| <code>getFrequencyOf(anEntry)</code> | $O(n)$ | $O(n)$ |
| <code>contains(anEntry)</code> | $O(1), O(n), O(n)$ | $O(1), O(n), O(n)$ |
| <code>toArray()</code> | $O(n)$ | $O(n)$ |
| <code>getCurrentSize(), isEmpty()</code> | $O(1)$ | $O(1)$ |

图 4-11 两种实现下 ADT 包操作的时间效率，用大 O 符号表示

如你所见，所有的操作在两种实现下都有相同的大 O 表示。这个现象是不寻常的，但它反映了 ADT 包的简单性。后面介绍的 ADT 中，至少会有某些操作，其时间效率会依其实现方式的不同而不同。

本章小结

- 算法的复杂度由算法运行时需要的时间和空间来描述。
- 算法的时间需求 $f(n)$ 具有最多 $g(n)$ 阶——即 $f(n)$ 是 $O(g(n))$ 的——如果存在正常数 c 和 N ，对于所有的 $n \geq N$ ，有 $f(n) \leq c \times g(n)$ 。
- 典型的增长率函数之间的关系如下：
 $1 < \log(\log n) < \log n < \log^2 n < n < n \log n < n^2 < n^3 < 2^n < n!$
- 定长数组实现和链式实现的 ADT 包操作的时间复杂度相同。这种情形是非典型的 ADT，但反映了源于包特性的实现细节。

练习

1. 使用大 O 符号，表示下列每个任务最差情况下的时间需求。描述你所做的任何假设。
 - a. 到达舞会后，向在场的每个人招手。
 - b. 房间里每个人和房间里的其他人招手。
 - c. 你爬上楼梯。
 - d. 你滑下楼梯栏杆。
 - e. 进入电梯后，按下按钮选择楼层。
 - f. 乘坐电梯从一层到 n 层。
 - g. 读一本书两遍。

2. 描述一种从楼梯底部爬上顶层，花费超过 $O(n^2)$ 时间的方法。
3. 使用大 O 符号，表示下列每个任务最差情况下的时间需求。
- 显示整数数组中的所有整数。
 - 显示结点链表中的所有整数。
 - 显示整数数组中的第 n 个整数。
 - 计算整数数组中前 n 个偶整数的和。
4. 使用大 O 定义，说明
- $6n^2+3$ 是 $O(n^2)$ 的
 - $n^2+17n+1$ 是 $O(n^2)$ 的
 - $5n^3+100 n^2-n-10$ 是 $O(n^3)$ 的
 - $3n^2+2^n$ 是 $O(2^n)$ 的
5. 算法 X 需要 n^2+9n+5 个操作，而算法 Y 需要 $5n^2$ 个操作。你能给出当 n 很小和 n 很大时，这两个算法的时间需求结论吗？这两种情况下哪个算法更快？
6. 说明对于 $a, b > 1$, $O(\log_a n) = O(\log_b n)$ 。提示： $\log_a n = \log_b n / \log_b a$ 。
7. 如果 $f(n)$ 是 $O(g(n))$ 的，而 $g(n)$ 是 $O(h(n))$ 的，使用大 O 定义，说明 $f(n)$ 是 $O(h(n))$ 的。
8. 段 4.9 及本章小结说明了典型增长率函数之间的关系。指明下列增长率函数在这个顺序中的位置：
- $n^2 \log n$
 - \sqrt{n}
 - $n^2 / \log n$
 - 3^n
9. 说明 $7n^2+5n$ 不是 $O(n)$ 的。
10. 下列计算的大 O 是多少？
- ```
int sum = 0;
for (int counter = n; counter > 0; counter = counter - 2)
 sum = sum + counter;
```
11. 下列计算的大  $O$  是多少？
- ```
int sum = 0;
for (int counter = 1; counter < n; counter = 2 * counter)
    sum = sum + counter;
```
12. 假定用 Java 语言如下实现某个算法：
- ```
for (int pass = 1; pass <= n; pass++)
{
 for (int index = 0; index < n; index++)
 {
 for (int count = 1; count < 10; count++)
 {
 .
 } // end for
 } // end for
} // end for
```

这个算法对含  $n$  项的数组进行处理。前面的代码只显示了算法中要重复的部分，但没有显示循环中的计算。不管怎样，这些计算与  $n$  无关。这个算法的阶是多少？

13. 将前一个练习的内层循环中的 10 用  $n$  替代，重做一遍。
14. `method1` 的大  $O$  是多少？它是最优情况和最差情况吗？

```
public static void method1(int[] array, int n)
{
 for (int index = 0; index < n - 1; index++)
```

```

{
 int mark = privateMethod1(array, index, n - 1);
 int temp = array[index];
 array[index] = array[mark];
 array[mark] = temp;
} // end for
} // end method1

public static int privateMethod1(int[] array, int first, int last)
{
 int min = array[first];
 int indexOfMin = first;
 for (int index = first + 1; index <= last; index++)
 {
 if (array[index] < min)
 {
 min = array[index];
 indexOfMin = index;
 } // end if
 } // end for
 return indexOfMin;
} // end privateMethod1

```

15. method2 的大  $O$  是多少？它是最优情况和最差情况吗？

```

public static void method2(int[] array, int n)
{
 for (int index = 1; index <= n - 1; index++)
 privateMethod2(array[index], array, 0, index - 1);
} // end method2

public static void privateMethod2(int entry, int[] array, int begin, int end)
{
 int index = end;
 while ((index >= begin) && (entry < array[index]))
 {
 array[index + 1] = array[index];
 index--;
 } // end while
 array[index + 1] = entry;
} // end privateMethod2

```

16. 考虑两个程序 A 和 B。程序 A 需要  $1000 \times n^2$  个操作，而程序 B 需要  $2^n$  个操作。 $n$  取什么值时，程序 A 将比程序 B 运行得快？

17. 考虑 4 个程序——A、B、C 和 D——有下列性能：

A:  $O(\log n)$

B:  $O(n)$

C:  $O(n^2)$

D:  $O(2^n)$

如果求解规模为 1000 的问题时每个程序需要 10 秒，评估求解规模为 2000 的问题时每个程序所需的时间。

18. 假定你有一个字典，其中的单词并没有按字典序排列。在这本字典中查找某个单词的时间复杂度是多少？表示为  $n$  的函数， $n$  是单词的个数。

19. 重做前一个关于字典的练习，其中的单词按字典序有序。比较你的结果与前一练习的结果。

20. 考虑一个足球运动员在足球场上奔跑冲刺。他从 0 码线开始，然后跑向 1 码线，然后跑回 0 码线。然后跑回 2 码线，再跑回 0 码线，跑向 3 码线，再返回 0 码线，以此类推，直到他到达 10 码线后返回 0 码线为止。

a. 他总共跑了多少码？

b. 如果他到达  $n$  码线而不是 10 码线，他总共跑了多少码？

c. 将他跑的总距离，与从 0 码线开始跑到  $n$  码线的运动员所跑的距离进行比较。

21. 考虑正整数序列  $A$  的下列定义：

$$A_{i+1} = \begin{cases} A_i / 2 & \text{如果 } A_i \text{ 是偶数} \\ 3A_i - 1 & \text{如果 } A_i \text{ 是奇数} \end{cases}$$

如果  $A_0$  是某个值  $v$ ，根据  $k$  和  $v$ ，给出  $A_k$  具有：

a. 最小值

b. 最大值

时的大  $O$  表示。

22. 第 2 章描述了 ADT 包使用定长数组的实现。对于 `add`、`remove` 和 `contains` 操作，哪个具有常数阶增长率函数？

23. 第 2 章描述了 ADT 包使用变长数组的实现。使用大  $O$  符号，导出段 2.41 中给出的 `doubleCapacity` 方法的时间复杂度。

24. 考虑长度为  $n$  的数组，按随机序保存  $1 \sim n+1$  之间的不重复整数。例如，长度为 5 的数组中，含有 5 个  $1 \sim 6$  中随机选出的不重复整数。所以，数组可能含有 3 6 5 1 4。整数  $1 \sim 6$  中，注意到 2 没被选中，所以不在数组中。

写 Java 代码，找到没有出现在这样的数组中的整数。你的方案应该使用

a.  $O(n^2)$  操作

b.  $O(n)$  操作

25. 考虑长度为  $n$  的数组，含有随机序的正、负整数。写 Java 代码，重排这些整数，让负整数出现在正整数之前。你的方案应该使用

a.  $O(n^2)$  操作

b.  $O(n)$  操作

## 项目

对于下列项目，你应该知道如何对一段 Java 代码进行计时。一种方法是使用 `java.util.Date` 类。`Date` 对象含有创建它时的时间。这个时间保存为一个 `long` 型整数，等于从格林尼治标准时间 1970 年 1 月 1 日 00:00:00.000 开始经过的毫秒数。结束时间毫秒数减去开始时间毫秒数，可以得到一段代码的运行时间——毫秒数。

例如，假定 `thisMethod` 是希望对它计时的方法名。下列语句将计算运行 `thisMethod` 所需要的毫秒数：

```
Date current = new Date(); // Get current time
long startTime = current.getTime();
thisMethod(); // Code to be timed
current = new Date(); // Get current time
long stopTime = current.getTime();
long elapsedTime = stopTime - startTime; // Milliseconds
```

1. 写 Java 程序，实现图 4-1 中的 3 个算法，并对不同的  $n$  值对算法计时。程序应该列表显示，对不同的  $n$ ，每个算法的运行时间。

2. 考虑下面两个循环：

```
// Loop A
for (i = 1; i <= n; i++)
 for (j = 1; j <= 10000; j++)
 sum = sum + j;

// Loop B
for (i = 1; i <= n; i++)
 for (j = 1; j <= n; j++)
 sum = sum + j;
```

虽然 Loop A 是  $O(n)$  的，而 Loop B 是  $O(n^2)$  的，但对于小的  $n$  值，Loop B 可以比 Loop A 更快。设计并实现一个实验，找到使 Loop B 更快的  $n$  值。

3. 重做前一项目，但使用下列 Loop B：

```
// Loop B
for (i = 1; i <= n; i++)
 for (j = 1; j <= n; j++)
 for (k = 1; k <= n; k++)
 sum = sum + k;
```

4. 第 2 章段 2.12 中给出了 ADT 包中方法 `toArray` 的定义，如下所示。

```
public T[] toArray()
{
 // The cast is safe because the new array contains null entries.
 @SuppressWarnings("unchecked")
 T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast
 for (int index = 0; index < numberOfEntries; index++)
 {
 result[index] = bag[index];
 } // end for
 return result;
} // end toArray
```

另一个替代定义是调用方法 `Arrays.copyOf`，如下所示：

```
public T[] toArray()
{
 return Arrays.copyOf(bag, bag.length);
} // end toArray
```

对于不同大小的包，比较这两个方法的运行时间。

5. 假定在台球桌上有几个带编号的球。每一步从桌上删除一个球。如果删除的球的编号是  $n$ ，则用  $n$  个  $n/2$  号球替换  $n$  号球，其中除法结果取整。例如，如果删除的是 5 号球，则用 5 个 2 号球来替代。写一个程序，模拟这个过程。使用正整数的包来表示台球桌上的球。

使用大  $O$  符号，预测当初始包仅含有值  $n$  时这个算法的时间需求。然后对于不同的  $n$  值，对程序的实际执行时间进行计时，并画出表示为  $n$  的函数的性能曲线。

6. 重做前一个项目，但将  $n$  号球替换为随机编号小于  $n$  的  $n$  个球。

7. 在神话中，九头蛇是个有很多头的怪兽。英雄每次砍下一个头，原位置就会长出两个更小的头。幸运的是，如果头足够小，英雄可以砍下它且不会在原地再生出另外两个。要杀死九头蛇，所有的英雄要做的就是砍下所有的头。

写一个程序模拟九头蛇。不是用头，而是使用字符串，即一个字符串包表示九头蛇。每次从包中删除一个字符串，删去字符串的首字符，并把剩余字符串的两个备份放回包中。例如，如果删除 HYDRA，则向包中添加两个 YDRA。如果删除单字符字，则什么也不放回包中。开始，从键盘读入一个字，将它放到空包中。当包变空时九头蛇死掉。

使用大  $O$  符号，预测初始字符串含  $n$  个字符时这个算法的时间需求。然后对于不同的  $n$  值，对程序的实际执行时间进行计时，并画出表示为  $n$  的函数的性能曲线。

8. 第 1 章的练习 5、练习 6 和练习 7 要求你分别规范说明 ADT 包中的 `union`、`intersection` 和 `difference` 方法。第 1 章项目 7 要求你仅使用 ADT 包操作定义这些方法。第 2 章练习 15、练习 16 和练习 17 要求你为类 `ResizableArrayBag` 规范定义这些方法，而第 3 章练习 9、练习 10 和练习 11 要求你为类 `LinkedBag` 规范定义这些方法。对于大的、随机生成的包，比较这 9 个方法的运行时间。

先修章节：序言、第1章、Java插曲2

### 目标

学习完本章后，应该能够

- 描述ADT栈的操作
- 使用栈来判定代数表达式中分隔符是否正确配对
- 使用栈将中缀表达式转为后缀表达式
- 使用栈来计算后缀表达式的值
- 使用栈来计算中缀表达式的值
- 在程序中使用栈
- 描述Java运行时环境如何使用栈来跟踪方法的执行过程

在日常生活中，栈是一个熟悉的事物。你或许见过桌上的一摞书，自助餐厅里的一摞盘子，毛巾柜中的一摞毛巾，或是阁楼中的一摞盒子。当你向栈中添加项时，会放在栈的顶上。当你移走一项时，拿走最上面的一个。这最上面的一个是最后放到栈中的。所以当我们移走一项时，移走的是最近添加进去的。即最后添加到栈中的项最先移走。

尽管我们的栈示例是这样的，可是日常生活中通常不表现为这种后进先出（Last-In First-Out, LIFO）行为。尽管最近被雇佣的职员常常是最先被解雇的，但我们生活在先来先服务的社会。不过，在计算机科学世界，后进先出恰恰是许多重要算法需要的行为。这些算法常用到抽象数据类型栈，这是表现为后进先出行为的ADT。例如，编译程序使用栈来解释代数表达式的含义，运行时环境使用栈来执行递归方法。

本章描述ADT栈，并提供几个使用示例。

## ADT栈的规范说明

ADT栈(stack)根据项的添加次序来组织项。所有的添加都位于称为栈顶(top)的一端。栈顶项(top entry)，即位于栈顶的项，是当前栈的项中最新的一项。图5-1展示了几个你熟悉的栈。

5.1

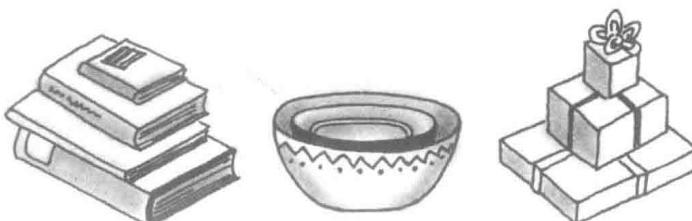


图5-1 几个熟悉的栈

注：当前栈的项中，最新添加的项在栈顶。（有些项或许是更靠后添加的，然后已被删除了。）

栈限制对其中项的访问。客户仅能看到或删除栈顶项。查看不在栈顶的项的唯一方法是，从栈中不停地删除项，直到要找的项到达栈顶。如果一个个地删除栈中所有的项，则得到反序的结果，最近进栈的在最前面，最先进栈的在最后面。

5.2 将项添加到栈中的操作常称为入栈（push）。删除操作称为出栈（pop）。获取栈顶项但不删除它的操作称为查看（peek）。一般地，不能在栈<sup>⊖</sup>中查找某个具体的项。下面的规范说明定义了 ADT 栈的一组操作。

| 抽象数据类型：栈       |                          |                                                   |
|----------------|--------------------------|---------------------------------------------------|
| 数据             |                          |                                                   |
| 操作             |                          |                                                   |
| 伪代码            | UML                      | 描述                                                |
| push(newEntry) | +push(newEntry: T): void | 任务：添加新项到栈顶<br>输入：newEntry 是新项<br>输出：无             |
| pop()          | +pop(): T                | 任务：删除并返回栈顶项<br>输入：无<br>输出：返回栈顶项。操作之前如果栈空，则抛出异常    |
| peek()         | +peek(): T               | 任务：获取栈顶项且不以任何方式改变栈<br>输入：无<br>输出：返回栈顶项。如果栈空，则抛出异常 |
| isEmpty()      | +isEmpty(): boolean      | 任务：检查栈是否为空<br>输入：无<br>输出：如果栈空，则返回真                |
| clear()        | +clear(): void           | 任务：从栈中删除所有项<br>输入：无<br>输出：无                       |



### 设计决策：当栈空时，pop 和 peek 应该如何做？

客户真的不应该在集合为空时，调用像 pop 和 peek 这样的方法，试图从集合中删除或是获取项。即便如此，在这种情形下这些方法的行为也必须合理。我们考虑这类方法的三种可能的动作：

- 假定集合不空；兑现一个前置条件来做这个假设。
- 返回 null。
- 抛出一个异常。

第一个选项适合于私有方法，因为它们仅被同一类内的其他方法调用。所以，你作为类的程序员，能够保证私有方法的前置条件在调用之前是满足的。私有方法转而又被它所信任的代码——即被同一类中的另一个方法——调用，所以可以保证前置条件是满足的。

栈方法 pop 和 peek 是公有方法；我们不能信任客户来兑现这些方法所需的任何前置条件。所以，第一个选项不是切实可行的选择。相反，这些方法必须假定栈可能为空，并预防这种情形。

<sup>⊖</sup> 不过 Java 类库中有一个定义了查找方法的栈类，本章后面会看到。

如果我们让方法返回一个值，用来代表某个问题，例如空集合，那么这个值必须具有方法的返回类型。因为没有从集合中删除项，所以返回 null 会很自然。只要集合的 ADT 中不允许含有 null 项，则这个选择就是合适的，因为 null 必须明确地表示集合中没有要返回的项。ADT 包就是这样的情形，所以它的 remove 方法返回 null 以表示不成功的操作。但是，我们想让 null 作为 ADT 栈中的合法项。所以 null 作为返回值的含义具有二义性：如果不额外调用 isEmpty，客户没办法知道 null 是栈中的数据项还是表示空栈的标志。需要客户调用第二个方法来解释另一个方法的动作，这或者导致对方法动作的误解，或者使得代码与前置条件处于相同的情形。注意，本书中的大多数 ADT 允许 null 值作为数据集中的合法项。

当栈为空时，我们只剩下抛出异常这一种选择。采用这种设计时，返回值 null 可认为是有效数据。



### 安全说明：信任

前面的设计决策中谈到了信任。你能信任一段代码吗？不能，除非你能证明它的动作正确且安全，这种情形下它成为可信代码 (trusted code)。你能信任客户以确定的方式使用你的软件，所以兑现任何及所有的前置条件，并能正确解释返回码吗？不能。但是类内的私有方法确实可以假设或信任其前置条件能被兑现，且它的返回值可被正确处理。



**设计决策：**当栈为空时，pop 和 peek 应该抛出哪类异常：受检异常还是运行时异常？一般地，如果方法的客户能在执行时从异常中合理地恢复，它就应该抛出受检异常。这种情况下，客户可以直接处理异常，或是将它传播到另一个方法中。另一方面，如果你把异常看作对你方法的不正常使用——即使用你方法的程序员的错误——则方法应该抛出运行时异常。运行时异常不需要——但可以——在 throws 子句中说明，而且也不需要——但可以——被客户捕获。

我们将栈为空时调用 pop 或 peek 方法看作客户的错误。所以抛出运行时异常。但如果应用程序能从这个事件中恢复，则它可以捕获这个异常并处理它。



### 注：方法的替代名字

类的设计者常常为某些方法再加个别名。例如，你可以在 ADT 栈中包含另外的方法 add 和 remove (或 insert 和 delete)，来表示 push 和 pop。另外，有时也用 pull 表示 pop，而 getTop 可用来表示 peek，所以将它们作为别名也是合理的。

程序清单 5-1 中的 Java 接口规范说明了对象的栈。泛型 T——它表示任意的类类型——是栈中项的数据类型。注意，EmptyStackException 是 Java 类库中 java.util 包中的运行时异常。

5.3

#### 程序清单 5-1 用于 ADT 栈的接口

```

1 public interface StackInterface<T>
2 {
3 /** Adds a new entry to the top of this stack.
4 * @param newEntry An object to be added to the stack. */
5 public void push(T newEntry);
6 }
```

```

7 /** Removes and returns this stack's top entry.
8 * @return The object at the top of the stack.
9 * @throws EmptyStackException if the stack is empty before
10 * the operation. */
11 public T pop();
12
13 /** Retrieves this stack's top entry.
14 * @return The object at the top of the stack.
15 * @throws EmptyStackException if the stack is empty. */
16 public T peek();
17
18 /** Detects whether this stack is empty.
19 * @return True if the stack is empty. */
20 public boolean isEmpty();
21
22 /** Removes all entries from this stack. */
23 public void clear();
24 } // end StackInterface

```

5.4  **示例：展示栈方法。**下列语句向 / 从栈中添加、获取及删除字符串。我们假定，类 OurStack 实现了 StackInterface 接口，且可供我们使用。

```

StackInterface<String> stringStack = new OurStack<>();
stringStack.push("Jim");
stringStack.push("Jess");
stringStack.push("Jill");
stringStack.push("Jane");
stringStack.push("Joe");

String top = stringStack.peek(); // Returns "Joe"
System.out.println(top + " is at the top of the stack.");
top = stringStack.pop(); // Removes and returns "Joe"
System.out.println(top + " is removed from the stack.");
top = stringStack.peek(); // Returns "Jane"
System.out.println(top + " is at the top of the stack.");
top = stringStack.pop(); // Removes and returns "Jane"
System.out.println(top + " is removed from the stack.");

```

图 5-2a ~ 图 5-2e 展示对栈的 5 次添加。此时，栈中——从栈顶到栈底——含有的字符串依次是 Joe、Jane、Jill、Jess 和 Jim。栈顶的字符串是 Joe；peek 操作可以得到它。方法 pop 再次得到 Joe，然后删除它（图 5-2f 所示）。接下来调用 peek 得到 Jane。之后 pop 得到 Jane 并删除它（图 5-2g 所示）。

再调用三次 pop 方法将删除 Jill、Jess 和 Jim，栈为空。后面的调用，不管是 pop 还是 peek，都会抛出 EmptyStackException 异常。

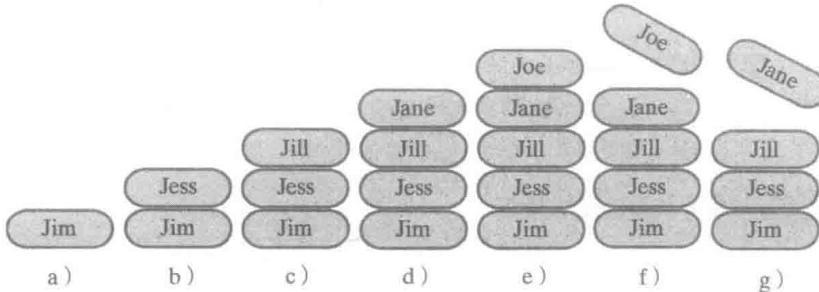


图 5-2 进行了下列操作后的字符串栈 a) push 添加了 Jim； b) push 添加了 Jess； c) push 添加了 Jill； d) push 添加了 Jane； e) push 添加了 Joe； f) pop 得到并删除 Joe； g) pop 得到并删除 Jane



### 安全说明：设计准则

- 使用前置条件和后置条件来说明假设。
- 不要信任客户能正确使用公有方法。
- 避免返回值的二义性。
- 宁愿抛出异常，也不要用返回值来表示一个问题。



### 学习问题 1 执行下列语句后，栈顶是哪个字符串？栈底是哪个字符串？

```
StackInterface<String> stringStack = new OurStack<>();
stringStack.push("Jim");
stringStack.push("Jazmin");
stringStack.pop();
stringStack.push("Sophia");
stringStack.push("Tia");
stringStack.pop();
```

### 学习问题 2 考虑学习问题 1 中创建的栈，定义新的空栈 nameStack。

- 写一个循环，从 stringStack 中出栈，并将结果入栈 nameStack 中。
- 描述当刚写的循环执行完毕，栈 stringStack 和 nameStack 中的内容。

## 使用栈来处理代数表达式

数学中，代数表达式由操作数和运算符组成，其中操作数是变量或常量，如 + 和 \* 这样的是运算符。我们使用 Java 中的符号 +、-、\* 和 / 分别表示加法、减法、乘法和除法。我们使用 ^ 来表示求幂，提示一下，Java 中没有求幂运算符；在 Java 中，^ 是异或运算符。

运算符一般有两个操作数，所以称为二元运算符 (binary operator)。例如， $a+b$  中的 + 是一个二元运算符。运算符 + 和 - 只有一个操作数时，它们也称为一元运算符 (unary operator)。例如， $-5$  中的负号是一个一元运算符。

当代数表达式中没有括号时，操作按确定的次序执行。最先求幂；它们的优先级 (precedence) 比其他的运算更高。接下来是乘法和除法运算，然后是加法和减法运算。例如，表达式

$$20 - 2 * 2 ^ 3$$

计算  $20 - 2 * 8$ ，然后是  $20 - 16$ ，最终得到 4。

但当两个或更多个相邻的运算符有相同的优先级时该如何呢？例如  $a ^ b ^ c$  中的求幂，是从右到左计算。所以  $2 ^ 2 ^ 3$  表示的是  $2 ^ {(2 ^ 3)}$ ，或是  $2^8$ ，而不是  $(2 ^ 2) ^ 3$ ，即  $4^3$ 。其他的运算符是从左至右计算，例如  $a * b / c$  中的乘法和除法，或是  $a - b + c$  中的加法和减法。所以， $8 - 4 + 2$  表示的是  $(8 - 4) + 2$ ，或 6，而不是  $8 - (4 + 2)$ ，即 2。表达式中的括号可以优先于运算符正常的优先级。

一般地，将二元运算符放在其操作数的中间，如在  $a + b$  中。这种形式熟悉的表达式称为中缀表达式 (infix expression)。其他的表示法也是可以的。例如，可以将二元运算符放在两个操作数之前。所以  $a + b$  变为  $+ ab$ 。这个表达式称为前缀表达式 (prefix expression)。或者可以将二元运算符放在两个操作数之后。所以  $a + b$  变为  $ab +$ 。这个表达式称为后缀表达式 (postfix expression)。虽然我们最熟悉中缀表达式，但前缀表达式和后缀表达式处理起来更简单，因为它们不使用优先级规则或是括号。前缀表达式或是后缀表达式中，运算符和操作数出现的次序，隐含表明了它们的优先级。本章后面将详细介绍这些表达式。



注：代数表达式

在中缀表达式中，每个二元运算符出现在它的操作数的中间。如  $a + b$  中。

在前缀表达式中，每个二元运算符出现在它的操作数的前面，如 $+ab$ 中。

在后缀表达式中，每个二元运算符出现在它的操作数的后面，如  $a\ b\ +$  中。



注：前缀表达式表示法有时称为波兰表示法（Polish notation），因为它由波兰数学家 Jan Lukasiewicz 于 20 世纪 20 年代提出。后缀表达式表示法有时称为逆波兰表示法（reverse Polish notation）。

问题求解：检查中缀代数表达式中平衡的分隔符



虽然程序员在 Java 中写代数表达式时使用圆括号，但是数学家出于同样的目的使用圆括号、方括号和花括号。这些分隔符必须正确配对。例如，一个开圆括号必须对应于一个闭圆括号。另外，分隔符对不能交叉。所以，表达式能够含有一系列分隔符，例如

{[0010]}

但不能含有

[1]

为方便起见，我们说，一个平衡表达式（balanced expression）包含配对正确或平衡的（balanced）分隔符。

我们想让一个算法来检测中缀表达式是否是平衡的。

56



示例：平衡表达式。我们来看看下面表达式是否是平衡的。

$$a \{ b [c(d+e)/2 - f] + 1 \}$$

从左至右扫描表达式，查看分隔符，忽略不是分隔符的其他任意符号。当遇到开分隔符时，必须保存它。当发现一个闭分隔符时，必须看看它是否对应于最近遇到的开分隔符。如果是，则丢掉开分隔符，并继续扫描表达式。如果能扫描完整个表达式且没有不匹配的情况，则表达式中的分隔符是平衡的。

能让我们保存对象然后获取或删除最近一个对象的 ADT 是栈。图 5-3 展示当我们扫描前面这个表达式时栈的内容。因为忽略所有非分隔符的符号，所以在这里将表达式表示为 `{[()]}{[()]}{[()]}` 就足够了。

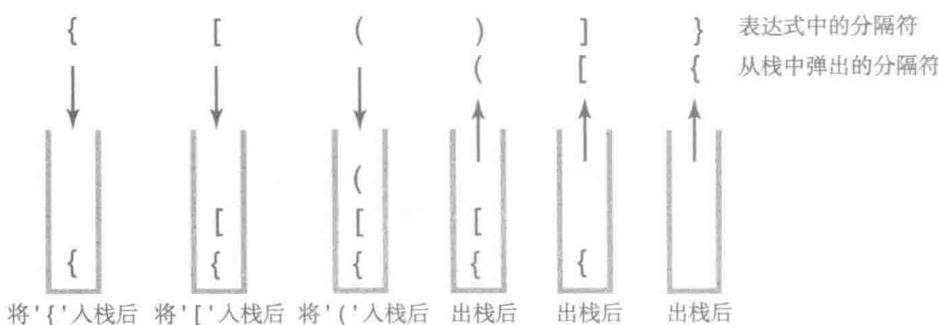


图 5-3 扫描含有平衡分隔符 { [ ( ) ] } 的表达式过程中栈的内容

将前三个开分隔符入栈后，开圆括号在栈顶。下一个分隔符是闭圆括号，它与栈顶的开圆括号配对。出栈，并继续比较闭方括号与目前栈顶的分隔符。它们也对应一致，所以再次出栈，并继续比较闭花括号与栈顶项。这两个分隔符是对应的，所以出栈。我们到达了表达式的末尾，而栈是空的。每个开分隔符正确对应到一个闭分隔符，所以分隔符是平衡的。

 **示例：不平衡的表达式。**让我们来检查某些含有不平衡分隔符的表达式。图 5-4 展示扫描含分隔符 { [ ( ) ] } 的表达式过程中栈的情况。5.7 这是有交叉分隔符对的例子。将前三个开分隔符入栈后，栈顶的开圆括号与表达式中下一个闭方括号不匹配。

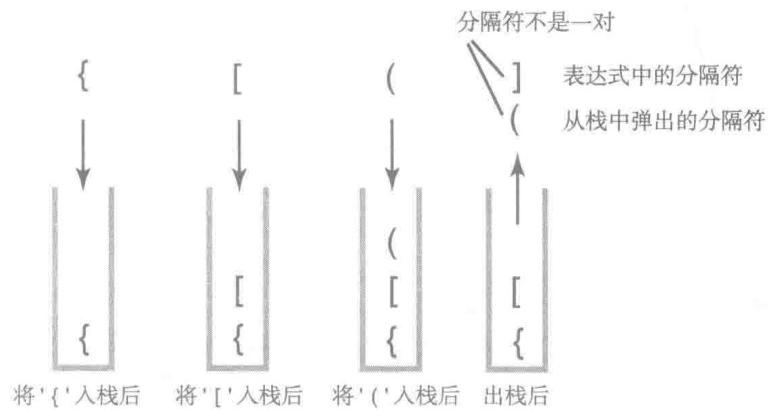


图 5-4 扫描含有不平衡分隔符 { [ ( ) ] } 的表达式过程中栈的内容

图 5-5 展示扫描含分隔符 [ ( ) ] } 的表达式过程中栈的情况。闭花括号没有对应的开花括号。当最终到达闭花括号时，栈为空。因为栈中不含有开花括号，故分隔符不是平衡的。

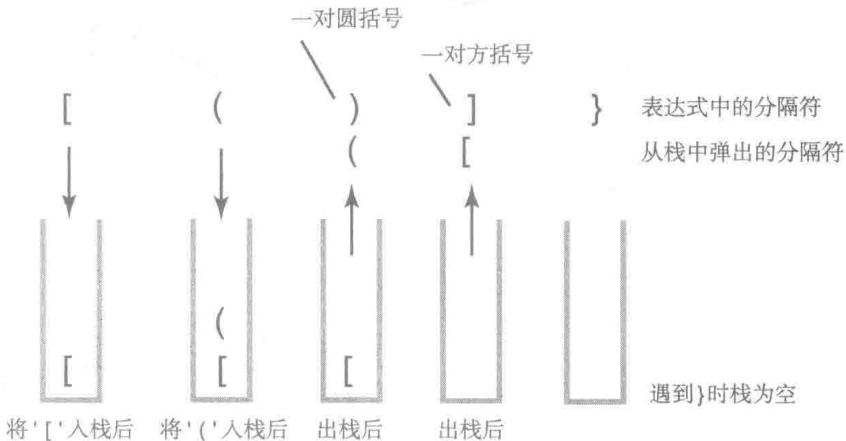


图 5-5 扫描含有不平衡分隔符 [ ( ) ] } 的表达式过程中栈的内容

图 5-6 展示扫描含分隔符 { [ ( ) ] } 的表达式过程中栈的情况。开花括号没有对应的闭花括号。当到达表达式尾时，已经处理了方括号和圆括号，栈中仍含有开花括号。因为遗留了这个分隔符，所以表达式含有不平衡的分隔符。

**算法。**前面的讨论和图提示了算法必须要处理的步骤。将这些结果形式化为下列伪代码：5.8

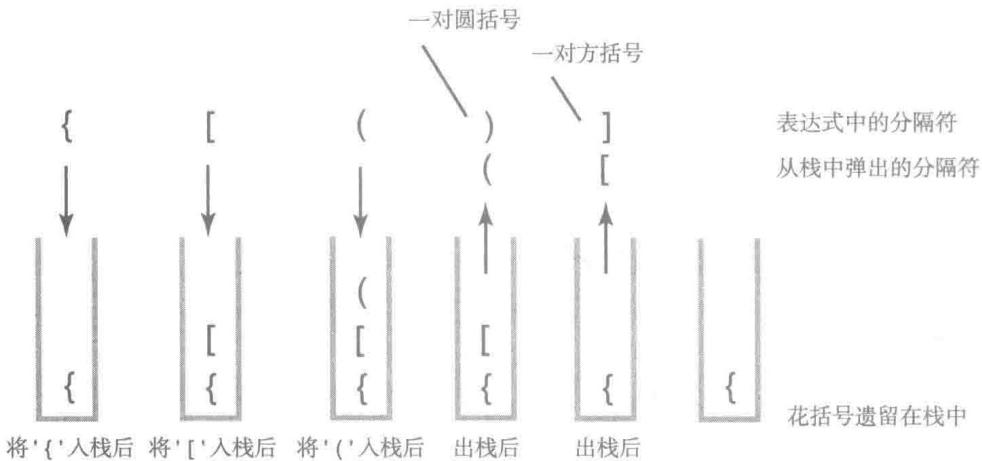


图 5-6 扫描含有不平衡分隔符 {[()]} 的表达式过程中栈的内容

```

Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true // The absence of delimiters is balanced
while ((isBalanced == true) 且没有到达 expression 尾)
{
 nextCharacter = expression 中的下一个字符
 switch (nextCharacter)
 {
 case '(': case '[': case '{':
 将 nextCharacter 入栈
 break
 case ')': case ']': case '}':
 if (栈空)
 isBalanced = false
 else
 {
 openDelimiter = 栈顶项
 出栈
 根据 openDelimiter 和 nextCharacter 是否是一对分隔符,
 决定 isBalanced = true 或 false
 }
 break
 }
 if (栈不空)
 isBalanced = false
 return isBalanced
}

```

**5.9** 使用前面各图中所给的每个示例，来检验这个算法。对于图 5-3 中的平衡表达式，当到达表达式结尾时 while 循环结束。栈为空，*isBalanced* 为真。对于图 5-4 中的表达式，当它发现闭方括号不能匹配开圆括号时，*isBalanced* 置为假，然后循环结束。实际上，栈不空不影响算法的结果。

对于图 5-5 中的表达式，遇到闭方括号时，因为栈是空的，故将标志 *isBalanced* 置为假，循环结束。最后，对于图 5-6 中的表达式，循环结束在表达式的末尾，此时 *isBalanced* 置为真。但栈不空——它含有一个开花括号——所以循环后，将 *isBalanced* 变为假。



**学习问题 3** 对于下列每个表达式，跟踪上边算法所给的 checkBalance，展示栈的内容。每种情形下 checkBalance 返回什么？

- $[a \{b / (c - d) + e/(f+g)\} - h]$
- $\{a [b + (c + 2)/d] + e\} + f\}$
- $[a \{b + [c (d + e) - f] + g\}]$

**Java 实现。**程序清单 5-2 中所示的类 BalanceChecker，将算法实现为一个静态方法 checkBalance。方法有一个参数，即作为字符串的表达式。假定类 OurStack 实现了 StackInterface，且是可用的。因为 StackInterface 规范说明的是对象栈，但前面的算法中使用的是字符栈，所以 checkBalance 使用包装类 Character 来创建适合于栈的对象。

### 程序清单 5-2 类 BalanceChecker

```

1 public class BalanceChecker
2 {
3 /** Decides whether the parentheses, brackets, and braces
4 * in a string occur in left/right pairs.
5 * @param expression A string to be checked.
6 * @return True if the delimiters are paired correctly. */
7 public static boolean checkBalance(String expression)
8 {
9 StackInterface<Character> openDelimiterStack = new OurStack<>();
10
11 int characterCount = expression.length();
12 boolean isBalanced = true;
13 int index = 0;
14 char nextCharacter = ' ';
15
16 while (isBalanced && (index < characterCount))
17 {
18 nextCharacter = expression.charAt(index);
19 switch (nextCharacter)
20 {
21 case '(': case '[': case '{':
22 openDelimiterStack.push(nextCharacter);
23 break;
24 case ')': case ']': case '}':
25 if (openDelimiterStack.isEmpty())
26 isBalanced = false;
27 else
28 {
29 char openDelimiter = openDelimiterStack.pop();
30 isBalanced = isPaired(openDelimiter, nextCharacter);
31 } // end if
32 break;
33 default: break; // Ignore unexpected characters
34 } // end switch
35 index++;
36 } // end while
37
38 if (!openDelimiterStack.isEmpty())
39 isBalanced = false;
40 return isBalanced;
41 } // end checkBalance
42
43 // Returns true if the given characters, open and close, form a pair
44 // of parentheses, brackets, or braces.
45 private static boolean isPaired(char open, char close)
46 {

```

```

47 return (open == '(' && close == ')') ||
48 (open == '[' && close == ']') ||
49 (open == '{' && close == '}');
50 } // end isPaired
51 } // end BalanceChecker

```

下列语句提供的示例说明了如何使用这个类：

```

String expression = "a {b [c (d + e)/2 - f] + 1}";
boolean isBalanced = BalanceChecker.checkBalance(expression);
if (isBalanced)
 System.out.println(expression + " is balanced");
else
 System.out.println(expression + " is not balanced");

```

### 问题求解：将中缀表达式转换为后缀表达式



我们最终的目标是展示如何计算中缀代数表达式，但后缀表达式更容易求值。所以我们先看看一个中缀表达式如何表示为后缀形式。

**5.11** 回忆后缀表达式中，二元运算符放在其两个操作数的后面。下面是几个中缀表达式及其对应的后缀形式的示例。

| 中缀            | 后缀              |
|---------------|-----------------|
| $a + b$       | $a\ b\ +$       |
| $(a + b) * c$ | $a\ b\ +\ c\ *$ |
| $a + b * c$   | $a\ b\ c\ *\ +$ |

注意到，操作数  $a$ 、 $b$  和  $c$  在中缀表达式中的次序，与在对应的后缀表达式中的次序相同。但是，运算符的次序不同了。这个次序依赖于运算符的优先级及括号的存在。正如我们提到过的，括号不出现在后缀表达式中。

**5.12** **手算策略。**可以从带完全括号的中缀表达式开始，来判定运算符将出现在后缀表达式中的什么地方。例如，将中缀表达式  $(a + b) * c$  写为  $((a + b) * c)$ 。通过添加括号，去掉了表达式对运算符优先级的依赖性。每个运算符现在都对应于一对括号。现在将每个运算符右移，紧贴在对应的闭括号的前面，得到  $((a\ b\ +)\ c\ *)$ 。最后，去掉括号，得到后缀表达式  $a\ b\ +\ c\ *$ 。

这个方法能让你理解后缀表达式中运算符的次序。当检验转换算法的结果时这也是有用的。不过，下面开发的算法不是基于这个思想的。



#### 学习问题 4 使用前一种方法，将下列每个中缀表达式转换为后缀表达式。

- a.  $a + b * c$
- b.  $a * b / (c - d)$
- c.  $a / b + (c - d)$
- d.  $a / b + c - d$

**5.13** **转换算法的基础。**为将中缀表达式转换为后缀形式，自左至右扫描中缀表达式。当遇到操作数时，将它放到正创建的新表达式的末尾。回忆一下，在中缀表达式中，操作数的次序与在对应的后缀表达式中是一样的。当遇到运算符时，必须先保存它，直到能判定它在所属的输出表达式中的位置时为止。例如，为转换中缀表达式  $a + b$ ，将  $a$  追加到初始为空的输出表达式尾，保存  $+$ ，再将  $b$  追加到输出表达式尾。现在需要获取  $+$ ，并将它追加到输出表

达式尾，得到后缀表达式  $a\ b\ +$ 。如果我们将运算符保存在栈中，则获取最近保存的运算符将非常容易。

在这个例子中，我们保存运算符，直到处理它的第二个操作数时为止。一般地，将运算符保存在栈中，至少要等到将它与下一个运算符的优先级进行比较时。例如，为转换表达式  $a + b * c$ ，将  $a$  追加到输出表达式尾，将  $+$  入栈，然后将  $b$  追加到输出中。现在根据下一个运算符  $*$  的优先级及栈顶  $+$  的优先级，来决定我们下一步的动作。因为  $*$  比  $+$  有更高的优先级，所以  $b$  不是加法的第二个操作数。而是加法要等待乘法的结果。所以将  $*$  入栈，操作数  $c$  追加到输出表达式尾。现在已经到达输入表达式的末尾了，此时从栈中弹出各个运算符，将其追加到输出表达式尾，得到后缀表达式  $a\ b\ c\ *\ +$ 。图 5-7 说明了这些步骤。栈显示为水平方向；最左元素是栈底。

| 中缀表达式的<br>下一个符号 | 后缀形式           | 运算符栈<br>( 栈底到栈顶 ) |
|-----------------|----------------|-------------------|
| $a$             | $a$            |                   |
| $+$             | $a$            | $+$               |
| $b$             | $a\ b$         | $+$               |
| $*$             | $a\ b$         | $+\ast$           |
| $c$             | $a\ b\ c$      | $+\ast$           |
|                 | $a\ b\ c\ *$   | $+$               |
|                 | $a\ b\ c\ * +$ |                   |

图 5-7 将中缀表达式  $a + b * c$  转换为后缀形式

**具有相同优先级的连续运算符。**如果连续的两个运算符有相同的优先级时怎么办？我们需要区分满足自左至右结合律的运算符——即  $+$ 、 $-$ 、 $*$  和  $/$ ——及求幂，后者满足自右至左结合律。例如，考虑表达式  $a - b + c$ 。当遇到  $+$  时，栈中含有运算符  $-$ ，且部分后缀表达式是  $a\ b$ 。减号运算符属于操作数  $a$  和  $b$ ，所以我们出栈，将  $-$  追加到表达式  $a\ b$  的后面。因为栈为空，故将  $+$  入栈。然后将  $c$  追加到结果中，最后出栈，追加  $+$ 。结果是  $a\ b\ -\ c\ +$ 。图 5-8a 说明了这些步骤。

现在来看表达式  $a \wedge b \wedge c$ 。遇到第二个求幂运算符时，栈中含有  $\wedge$ ，而到目前为止的结果是  $a\ b$ 。与之前一样，当前运算符与栈顶项有相同的优先级。但因为  $a \wedge b \wedge c$  的含义是  $a \wedge (b \wedge c)$ ，所以必须将第二个  $\wedge$  入栈，如图 5-8b 所示。

| 中缀表达式的<br>下一个符号 | 后缀形式            | 运算符栈<br>( 栈底到栈顶 ) |
|-----------------|-----------------|-------------------|
| $a$             | $a$             |                   |
| $-$             | $a$             | $-$               |
| $b$             | $a\ b$          | $-$               |
| $+$             | $a\ b\ -$       | $+$               |
| $c$             | $a\ b\ -\ c$    | $+$               |
|                 | $a\ b\ -\ c\ +$ |                   |

图 5-8 将中缀表达式转换为后缀形式

b)  $a \wedge b \wedge c$

| 中缀表达式的<br>下一个符号 | 后缀形式                 | 运算符栈<br>( 栈底到栈顶 ) |
|-----------------|----------------------|-------------------|
| $a$             | $a$                  | $\wedge$          |
| $\wedge$        | $a$                  |                   |
| $b$             | $a b$                | $\wedge$          |
| $\wedge$        | $a b$                | $\wedge\wedge$    |
| $c$             | $a b c$              | $\wedge\wedge$    |
|                 | $a b c \wedge$       | $\wedge$          |
|                 | $a b c \wedge\wedge$ |                   |

图 5-8 (续)

### 学习问题 5 一般的，应该在什么时候将求幂运算符 $\wedge$ 入栈？



5.15

**圆括号。**圆括号优先于运算符优先级规则。我们总是将开圆括号入栈。一旦它在栈中，我们就将开圆括号看作有最低优先级的运算符。即随后的一个任意的运算符都将入栈。当遇到闭圆括号时，将运算符出栈，且追加到已得到的后缀表达式尾，直到弹出一个开圆括号时为止。算法继续，但不将圆括号追加到后缀表达式中。



### 注：中缀到后缀的转换

为将中缀表达式转为后缀形式，在自左至右处理中缀表达式的过程中，根据遇到的符号，需要采取下列步骤：

- 操作数 将每个操作数追加到输出表达式尾。
- 运算符  $\wedge$  将  $\wedge$  入栈。
- 运算符 +、-、\* 或 / 将运算符出栈，将它们追加到输出表达式尾，直到栈空或是弹出项比新运算符的优先级更低。然后将新运算符入栈。
- 开圆括号 将 ( 入栈。
- 闭圆括号 将运算符出栈，将它们追加到输出表达式尾，直到弹出开圆括号。丢掉两个圆括号。

5.16

**中缀到后缀转换算法。**下列算法包含了前面对转换过程的讨论结果。为简化起见，表达式中的所有操作数都是单字符变量。

```

Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.

operatorStack = 新的空栈
postfix = 新的空字符串
while (infix还有待解析的符号)
{
 nextCharacter = infix 中下一个非空字符
 switch (nextCharacter)
 {
 case 变量:
 将 nextCharacter 追加到 postfix
 break
 case '^':
 operatorStack.push(nextCharacter)
 break
 }
}

```

```

case '^' :
 operatorStack.push(nextCharacter)
 break

case '+' : case '-' : case '*' : case '/' :
 while (!operatorStack.isEmpty() 且
 nextCharacter 的优先级 <= operatorStack.peek()) 的优先级
 {
 将 operatorStack.peek() 追加到 postfix
 operatorStack.pop()
 }
 operatorStack.push(nextCharacter)
 break

case '(' :
 operatorStack.push(nextCharacter)
 break

case ')' : // Stack is not empty if infix expression is valid
 topOperator = operatorStack.pop()
 while (topOperator != '(')
 {
 将 topOperator 追加到 postfix
 topOperator = operatorStack.pop()
 }
 break
default: break // Ignore unexpected characters
}
}

while (!operatorStack.isEmpty())
{
 topOperator = operatorStack.pop()
 将 topOperator 追加到 postfix
}
return postfix

```

图 5-9 跟踪了这个算法处理中缀表达式  $a / b * (c + (d - e))$  时的过程。得到的后缀表达式是  $a b / c d e - + *$ 。

| 中缀表达式的<br>下一个符号 | 后缀形式                | 运算符栈<br>( 栈底到栈顶 ) |
|-----------------|---------------------|-------------------|
| $a$             | $a$                 |                   |
| $/$             | $a$                 | $/$               |
| $b$             | $a b$               | $/$               |
| $*$             | $a b /$             |                   |
| $($             | $a b /$             | $*$               |
| $c$             | $a b / c$           | $* ($             |
| $+$             | $a b / c$           | $* ( +$           |
| $($             | $a b / c$           | $* (+ ($          |
| $d$             | $a b / c d$         | $* (+ ( -$        |
| $-$             | $a b / c d$         | $* (+ ( -$        |
| $e$             | $a b / c d e$       | $* (+ ( -$        |
| $)$             | $a b / c d e -$     | $* (+ ( -$        |
|                 | $a b / c d e -$     | $* (+$            |
|                 | $a b / c d e - +$   | $* ($             |
|                 | $a b / c d e - +$   | $*$               |
|                 | $a b / c d e - + *$ |                   |

图 5-9 将中缀表达式  $a / b * (c + (d - e))$  转换为后缀形式的步骤



**学习问题 6** 使用前一个算法，将下列每个中缀表达式表示为后缀表达式：

- $(a + b) / (c - d)$
- $a / (b - c) * d$
- $a - (b / (c - d) * e + f) ^ g$
- $(a - b * c) / (d * e ^ f * g + h)$

### 问题求解：计算后缀表达式的值



计算使用运算符 +、-、\*、/ 和 ^ 表示加法、减法、乘法、除法和求幂的后缀表达式的值。

5.17

计算后缀表达式不需要运算符优先级规则，因为运算符和操作数的次序已表明了运算的次序。另外，后缀表达式中不含有让计算复杂化的圆括号。

当扫描后缀表达式时，必须保存操作数，直到发现应用于它们的运算符时为止。例如，在计算后缀表达式  $a\ b\ /\$  时，找到变量  $a$  和  $b$  的存储位置，并保存它们的值<sup>⊖</sup>。当遇到运算符 / 时，它的第二个操作数是最近保存的值——即  $b$  的值。之前保存的值—— $a$  的值——是运算符的第一个操作数。将值保存在栈中，能让我们按需访问用于运算符的操作数。图 5-10 跟踪了当  $a$  是 2 且  $b$  是 4 时，计算  $a\ b\ /\$  值的过程。结果 0 表示是整除操作。

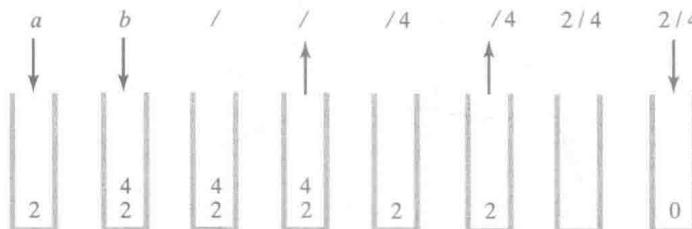


图 5-10 当  $a$  是 2 且  $b$  是 4 时，计算后缀表达式  $a\ b\ /\$  时的栈

现在考虑后缀表达式  $a\ b\ +\ c\ /$ ，这里  $a$  是 2， $b$  是 4 而  $c$  是 3。该表达式对应的中缀表达式是  $(a + b) / c$ ，所以它的值应该是 2。找到变量  $a$  后，将它的值 2 入栈。类似地，将  $b$  的值 4 入栈。下一个运算是 +，所以从栈中弹出两个值，让它们相加，将和值 6 入栈。注意，这个和值将是 / 运算符的第一个操作数。变量  $c$  是后缀表达式中的下一个符号，所以将它的值 3 入栈。最后，遇到运算符 /，这样从栈中弹出两个值，得到它们的商 6/3。将这个结果入栈。现在位于表达式的末尾了，单一的值 2 在栈中。这个值是表达式的值。图 5-11 跟踪了这个后缀表达式的计算过程。

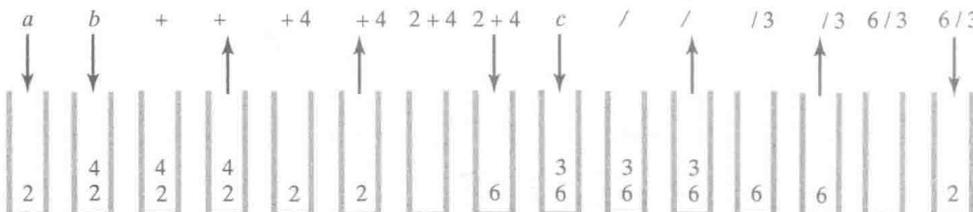


图 5-11 当  $a$  是 2、 $b$  是 4 且  $c$  是 3 时，计算后缀表达式  $a\ b\ +\ c\ /$  时的栈

<sup>⊖</sup> 找到变量的值不是项简单的任务，但本书不研究这个细节。

由这些示例直接得到下列求值算法。

```
Algorithm evaluatePostfix(postfix)
 // Evaluates a postfix expression.

 valueStack = 一个新空栈
 while (postfix还有待解析的符号)
 {
 nextCharacter = postfix 中下一个非空字符
 switch (nextCharacter)
 {
 case 变量:
 valueStack.push(变量 nextCharacter 的值)
 break
 case '+' : case '-' : case '*' : case '/' : case '^' :
 operandTwo = valueStack.pop()
 operandOne = valueStack.pop()
 result = nextCharacter 中的运算符与其操作数
 operandOne 和 operandTwo 的运算结果
 valueStack.push(result)
 break
 default: break // Ignore unexpected characters
 }
 }
 return valueStack.peek()
```

可以将这个算法和段 5.16 中给出的 `convertToPostfix`, 都实现为类 `Postfix` 的静态方法。具体实现留作练习。



**学习问题 7** 使用前一个算法, 计算下列每个后缀表达式的值。假定  $a = 2$ ,  $b = 3$ ,  $c = 4$ ,  $d = 5$  且  $e = 6$ 。

- $a\ e + b\ d - /$
- $a\ b\ c * d * -$
- $a\ b\ c - / d *$
- $e\ b\ c\ a ^ * + d -$

### 问题求解：计算中缀表达式的值



计算使用运算符 +、-、\*、/ 和 ^ 表示加法、减法、乘法、除法和求幂的中缀表达式的值。

使用段 5.16 和段 5.18 中的两个算法, 先将中缀表达式转换为等价的后缀表达式, 然后再求值, 就可以计算中缀表达式的值。但可以将两个算法合为一个算法, 使用两个栈直接计算中缀表达式的值, 就可以省去一些中间过程。这个合并算法根据将中缀表达式转为后缀形式的算法, 来维护一个运算符栈。但该算法不将操作数追加到表达式的末尾, 而是根据计算后缀表达式值的算法, 将操作数的值压入第二个栈中。



示例。考虑中缀表达式  $a + b * c$ 。当  $a$  是 2、 $b$  是 3 且  $c$  是 4 时, 表达式的值是 14。为计算这个结果, 将变量  $a$  的值入值栈, 将 + 入运算符栈, 将  $b$  的值入值栈。因为 \* 比运算符栈顶的 + 有更高的优先级, 所以将它入栈。最后, 将  $c$  的值入值栈。图 5-12a 显示了此时两个栈的状态。

现在弹出运算符栈，得到 $*$ 。通过弹出值栈两次，分别得到这个运算符的第二操作数和第一操作数。在计算乘积 $3 * 4$ 后，将结果12入值栈，如图5-12b所示。类似的情况，弹出运算符栈一次，弹出操作数栈两次，计算 $2 + 12$ ，将结果14入值栈。因为现在运算符栈为空，故值栈的栈顶即是表达式的值14。图5-12c显示了最后的这几步。

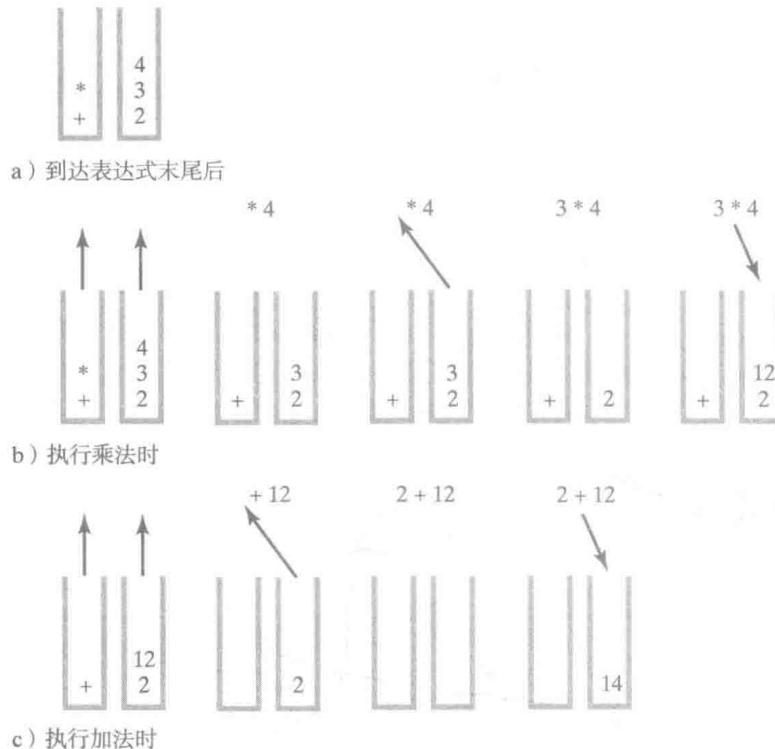


图5-12 当 $a$ 是2、 $b$ 是3且 $c$ 是4时计算 $a+b*c$ 过程中的两个栈

### 5.21

算法。计算中缀表达式的算法如下所示。你应该能从之前的算法中明白它的逻辑。

```

Algorithm evaluateInfix(infix)
// Evaluates an infix expression.

operatorStack = 一个新的空栈
valueStack = 一个新的空栈
while (infix 还有待处理的符号)
{
 nextCharacter = infix 中下一个非空字符
 switch (nextCharacter)
 {
 case 变量:
 valueStack.push(变量 nextCharacter 的值)
 break
 case '^' :
 operatorStack.push(nextCharacter)
 break
 case '+' : case '-' : case '*' : case '/' :
 while (!operatorStack.isEmpty() 且
 nextCharacter 的优先级 <= operatorStack.peek() 的优先级)
 {
 // Execute operator at top of operatorStack
 topOperator = operatorStack.pop()
 operandTwo = valueStack.pop()
 operandOne = valueStack.pop()
 }
 }
}

```

```

 result = topOperator 中的运算符与其操作数
 operandOne 和 operandTwo 的运算结果
 valueStack.push(result)
 }
 operatorStack.push(nextCharacter)
 break
 case '(' :
 operatorStack.push(nextCharacter)
 break
 case ')' : // Stack is not empty if infix expression is valid
 topOperator = operatorStack.pop()
 while (topOperator != '(')
 {
 operandTwo = valueStack.pop()
 operandOne = valueStack.pop()
 result = topOperator 中的运算符与其操作数
 operandOne 和 operandTwo 的运算结果
 valueStack.push(result)
 topOperator = operatorStack.pop()
 }
 break
 default: break // Ignore unexpected characters
}
}
while (!operatorStack.isEmpty())
{
 topOperator = operatorStack.pop()
 operandTwo = valueStack.pop()
 operandOne = valueStack.pop()
 result = topOperator 中的运算符与其操作数
 operandOne 和 operandTwo 的运算结果
 valueStack.push(result)
}
return valueStack.peek()

```



**学习问题 8** 使用前一个算法，计算下列每个中缀表达式的值。假定  $a = 2$ ,  $b = 3$ ,  
 $c = 4$ ,  $d = 5$  且  $e = 6$ 。

- a.  $a + b * c - 9$
- b.  $(a + e) / (b - d)$
- c.  $a + (b + c * d) - e / 2$
- d.  $e - b * c ^ a + d$

## 程序栈

当程序执行时，称为程序计数器（program counter）的一个特殊内存位置指向当前指令。5.22  
 程序计数器可能是实际计算机的一部分，或者，在 Java 中，是虚拟计算机的一部分<sup>⊖</sup>。

当调用方法时，程序运行时环境为方法创建一个称为活动记录（activation record）或框架（frame）的对象。活动记录显示运行期间方法的状态。具体来说，活动记录中含有方法的实参、局部变量和指向当前指令的引用——即程序计数器的副本。调用方法时，活动记录入栈，这个栈称为程序栈（program stack），或在 Java 中称为 Java 栈（Java stack）。因为一个方法可以调用另一个方法，故程序栈中常常含有多个活动记录。栈顶的记录属于当前正在运

<sup>⊖</sup> 为保持计算机的独立性，Java 代码运行在称为 Java 虚拟机（Java virtual Machine, JVM）的虚拟计算机上。

行的方法。刚刚压在栈顶下面的记录属于调用当前方法的方法，等等。

`main` 方法调用 `methodA`，后者又调用 `methodB` 时的程序栈如图 5-13 所示。当 `main` 开始运行时，它的活动记录位于程序栈的栈顶（图 5-13a）。当 `main` 调用 `methodA` 时，新的记录入栈。此时程序计数器是 50。图 5-13b 展示了修改后的 `main` 的记录，及刚开始执行的新的 `methodA` 的记录。当 `methodA` 调用 `methodB` 时，程序计数器是 120。新的活动记录入栈。图 5-13c 展示了修改后的 `main` 的记录、修改后的 `methodA` 的记录，及刚开始执行的新 `methodB` 的记录。

```

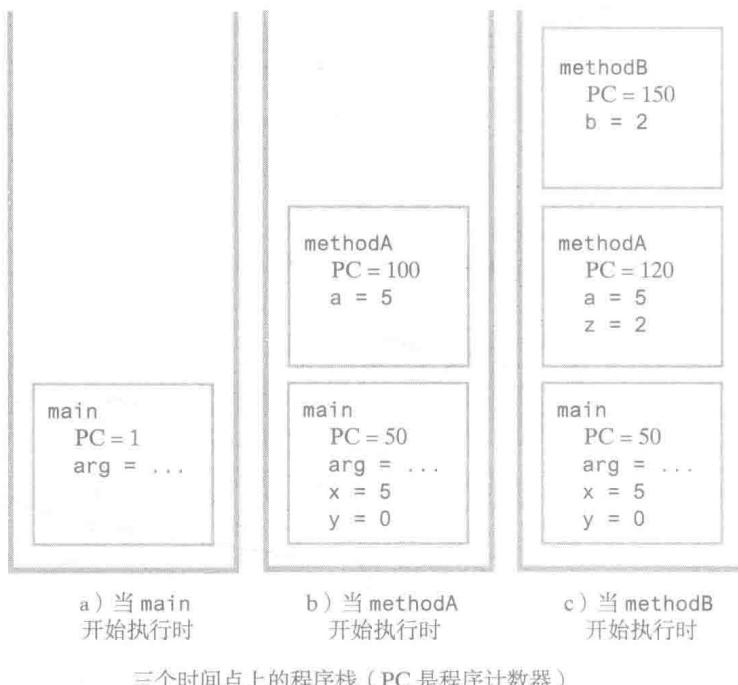
1 public static
void main(string[] arg)
{
 .
 .
 int x = 5;
50 int y = methodA(x);
 .
} // end main

100 public static
int methodA(int a)
{
 .
 .
 int z = 2;
120 methodB(z);
 .
 return z;
} // end methodA

150 public static
void methodB(int b)
{
 .
} // end methodB

```

程序



三个时间点上的程序栈 (PC 是程序计数器)

图 5-13 程序执行时的程序栈

当 `methodB` 执行时，它的活动记录被更新了，但 `main` 和 `methodA` 的记录没有改变。例如，`methodA` 的记录体现的是调用 `methodB` 的那个时刻本方法的状态。当 `methodB` 执行完毕，它的记录从栈中弹出。程序计数器重置回 120，然后进到下一条指令。所以 `methodA` 使用其活动记录中所给的实参和局部变量的值恢复执行。最终，`methodA` 完成执行，它的活动记录也从程序栈中弹出，`main` 继续执行直到完成。

## Java 类库：类 Stack

**5.23** Java 类库含有类 `Stack`，它实现了 `java.util` 包中的 ADT 栈。这个类仅有一个构造方法——创建一个空栈的默认构造方法。另外，类中的下面 4 个方法类似于我们在 `Stack-Interface` 中定义的方法。与我们定义的方法的不同之处已做标记。

```

public T push(T item);
public T pop();
public T peek();
public boolean empty();

```

`Stack` 中还定义了能让你查找或遍历栈中项的方法，以及传统的 ADT 栈中不支持的其

他方法。

 注：标准类 `java.util.Stack` 对栈的实现，比一些新的标准类中所提供的更慢。正如你在第 7 章将看到的，当你不想定义自己的栈类时，应该使用实现了接口 `java.util.Deque` 的类——比如 `ArrayDeque`。但是，现在可以使用 `Stack`，我们将在下一章定义自己的栈类。

## 本章小结

- ADT 栈按后进先出的原则组织项。栈顶的项是最新添加进来的。
- 栈的主要操作——`push`、`pop` 和 `peek`——都仅处理栈顶。方法 `push` 将项添加到栈顶；`pop` 删除并返回栈顶，而 `peek` 只是返回栈顶。
- 有两个操作数的算术运算符是二元运算符。当像 `+` 或 `-` 这样的运算符有一个操作数时，它是一元运算符。
- 代数表达式常含有圆括号、方括号和花括号。可以使用栈来检查这些分隔符是否正确配对。
- 普通的代数表达式称为中缀表达式，因为每个二元运算符出现在它的两个操作数的中间。中缀表达式需要运算符优先级规则，且可使用圆括号优先于这些规则。
- 在后缀表达式中，每个二元运算符出现在它的两个操作数的后面。在前缀表达式中，每个二元运算符出现在它的两个操作数的前面。后缀表达式和前缀表达式中不使用圆括号，且没有运算符优先级规则。
- 对给定的中缀表达式，生成等价的后缀表达式时，可以使用运算符栈。
- 可以使用值栈来计算后缀表达式的值。
- 可以使用两个栈——一个用于运算符，一个用于值——来计算中缀表达式的值。
- 当调用方法时，Java 运行时环境创建一个活动记录或框架，来记录方法的状态。记录中含有方法的实参和局部变量，还有当前指令的地址。记录放到称为程序栈的栈中。

## 程序设计技巧

- 像 `peek` 和 `pop` 这样的方法，当栈为空时必须有合理的动作。例如，它们可以返回 `null` 或是抛出一个异常。

## 练习

1. 如果将对象 `x`、`y` 和 `z` 压入初始为空的栈中，执行 3 次连续的 `pop` 操作，会以什么顺序将它们从栈中删除？
2. 创建含 3 个字符串 "Carlos"、"Darius" 和 "Sophia" 的栈，而 "Carlos" 位于栈顶的伪代码语句是什么？
3. 假定 `s` 和 `t` 是空栈，而 `a`、`b`、`c` 和 `d` 都是对象。执行下列操作序列后，得到的栈是什么？

```
s.push(a);
s.push(b);
s.push(c);
t.push(d);
t.push(s.pop());
t.push(s.peek());
```

```
s.push(t.pop());
t.pop();
```

4. 执行下列语句后，栈 pile 的内容是什么？假定 MyStack 是实现了接口 StackInterface 的类。

```
StackInterface<String> pile = new MyStack<>();
pile.push("Jazmin");
pile.push("Jess");
pile.push("Jack");
pile.push(pile.pop());
pile.push(pile.peek());
pile.push("Seiji");
String name = pile.pop();
pile.push(pile.peek());
```

5. 考虑下列 Java 语句，假定 MyStack 是实现了接口 StackInterface 的类：

```
int n = 4;
StackInterface<Integer> stack = new MyStack<>();
while (n > 0)
{
 stack.push(n);
 n--;
} // end while

int result = 1;
while (!stack.isEmpty())
{
 int integer = stack.pop();
 result = result * integer;
} // end while
System.out.println("result = " + result);
```

- a. 当执行这段代码时将显示什么值？
  - b. 这段代码计算的是哪个数学函数的值？
6. 对下列每个表达式，跟踪段 5.8 给出的算法 checkBalance 时，栈的内容是什么？
- a.  $a \{b [c * (d + e)] - f\}$
  - b.  $\{a (b * c) / [d + e] / f\} - g$
  - c.  $a \{b [c - d] e\} f$
7. 使用段 5.16 所给的算法 convertToPostfix，将下列每个中缀表达式转换为后缀表达式：
- a.  $a * b / (c - d)$
  - b.  $(a - b * c) / (d * e * f + g)$
  - c.  $a / b * (c + (d - e))$
  - d.  $(a ^ b * c - d) ^ e + f ^ g ^ h$
8. 使用段 5.18 所给的算法 evaluatePostfix，计算下列每个后缀表达式的值。假定  $a = 2$ ,  $b = 3$ ,  $c = 4$ ,  $d = 5$  且  $e = 6$ 。
- a.  $a b + c * d -$
  - b.  $a b * c a - / d e * +$
  - c.  $a c - b ^ d +$
9. 前一个练习中所给的各后缀表达式所对应的中缀表达式分别是什么？
10. 当跟踪段 5.21 所给的算法 evaluateInfix，计算下列每个中缀表达式时，展示两个栈的内容。假定  $a = 2$ ,  $b = 3$ ,  $c = 4$ ,  $d = 5$ ,  $e = 6$  且  $f = 7$ 。
- a.  $(a + b) / (c - d) - 5$
  - b.  $(d * f + 1) * e / (a ^ b - b * c + 1) - 72$
  - c.  $(a ^ c - f) ^ a - a ^ b ^ a$
11. 回文 (palindrome) 是一个符号串（一个单词、一个短语或是一个句子），不管向前读还是向后读都

是一样的——假定忽略空格、标点和大小写。例如，Race car 是一个回文。A man, a plan, a canal: Panama 也是。描述如何使用栈来测试一个串是否是回文。

12. 假定读入一个二进制串——由 0 和 1 组成的字符串——一次读入一个符号。描述如何使用栈但不是用计算的方法，来看 0 的个数是否等于 1 的个数。当它们的个数不等时，如何说明哪个符号——0 或 1——更多些，且比另一个多多少？
13. 写 Java 代码，按其入栈的次序显示栈中的所有对象。显示完所有对象后，栈应该与开始时有相同的内容。
14. 重新设计 ADT 包，让包可以含有 null 项。修改第 1 章程序清单 1-1 中的 BagInterface，让其对新设计有所反映。

## 项目

1. 使用类 `java.util.Stack`，定义实现了程序清单 5-1 中所给的接口 `StackInterface` 的类 `OurStack`。
2. 使用前一个项目中的 `OurStack`，编写验证程序清单 5-2 中所给的 `BalanceChecker` 类的程序。接下来的项目中当必须使用栈时，都要求你使用项目 1 中所定义的类 `OurStack`。
3. 写 Java 程序，使用栈来测试输入的字符串是否为回文。练习 11 中定义了“回文”，并要求你描述这个问题的解决方案。
4. 定义类 `Postfix`，它包括静态方法 `convertToPostfix` 和 `evaluatePostfix`。这些方法应该分别实现段 5.16 和段 5.18 所给的算法。假定给定的代数表达式的语法是正确的。Java 类库中的标准类 `StringBuilder`，及补充材料 1（在线）的段 S1.79 中所描述的内容对你完成项目会有帮助。
5. 使用段 5.21 所给的算法定义计算中缀表达式值的方法，并进行演示。假设表达式的语法是正确的，且使用单字符操作数。
6. 重复前一个项目，但去掉表达式语法正确这个假设条件。
7. 在 Lisp 语言中，4 个基本算术运算符之一出现在任意多个由空格隔开的操作数之前。得到的表达式括在圆括号中。运算符的动作如下：
  - $(+ a b c \dots)$  返回所有操作数的和，而  $(+)$  返回 0。
  - $(- a b c \dots)$  返回  $a - b - c - \dots$ ，而  $(- a)$  返回  $-a$ 。减号运算符必须至少有一个操作数。
  - $(* a b c \dots)$  返回所有操作数的乘积，而  $(*)$  返回 1。
  - $(/ a b c \dots)$  返回  $a / b / c / \dots$ ，而  $(/ a)$  返回  $1/a$ 。除法运算符必须至少有一个操作数。
 可以使用全括号的前缀形式，将这些基本表达式用圆括号括起来，组成更大的代数表达式。例如，下列是合法的 Lisp 表达式：

```
(+ (- 6) (* 2 3 4) (/ (+ 3) (*) (- 2 3 1)))
```

这个表达式的计算如下所示：

```
(+ (- 6) (* 2 3 4) (/ 3 1 -2))
(+ -6 24 -1.5)
16.5
```

设计并实现一个算法，使用栈来计算由 4 种基本运算符和整型值组成的合法的 Lisp 表达式的值。写一个程序读入这样的表达式，并验证你的算法。

8. 考虑前一个项目中所描述的代数表达式。操作数允许是整型值或是由字符串表示的变量名。设计并实现一个迭代算法，使用栈来测试一个表达式是否是 Lisp 中的合法表达式。写程序读入可能的表达式并验证你的算法。

你的程序读入的每个表达式可以分散在几行中，这是典型的 Lisp 程序员使用的风格。例如，下列表达式在 Lisp 中是合法的：

```
(+ (- height)
 (* 3 3 4)
 (/ 3 width length)
 (* radius radius)
)
```

相反，下列表达式在 Lisp 中是不合法的：

|                                                                              |                                                                                            |                                                                                       |                                                                                        |
|------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| $(+ (-$<br>$(* 3 3 4)$<br>$(/ 3 width length)$<br>$(* radius radius)$<br>$)$ | $(+ (- height)$<br>$(* 3 3 4) )s$<br>$(* (/ 3 width length)$<br>$(* radius radius)$<br>$)$ | $(+ (- height)$<br>$(* 3 3 4)$<br>$(/ 3 width length))$<br>$(* radius radius)$<br>$)$ | $(+ (- height)$<br>$(* 3 3 4)$<br>$((/ 3 width length))$<br>$(* radius radius)$<br>$)$ |
|------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|

9. 写一个程序，图示化显示用于简单中缀表达式的可用的计算器；表达式由单数字操作数，+、-、\*、/ 运算符及圆括号组成。有下列假定：

- 一元运算符（如 -2 中的）是不合法的。
- 包除法在内的所有操作都是整数操作。
- 输入的表达式中不能含有嵌入的空格和其他不合法的字符，因为它是用小键盘输入的。
- 输入的表达式是语法正确的中缀表达式。
- 不会出现被 0 除现象。（考虑如何去掉这条限制。）

计算器有一个显示窗口和一个含 20 个键的键盘，布局如下所示：

```
C < Q /
7 8 9 *
4 5 6 -
1 2 3 +
0 () =
```

当使用者按键键入中缀表达式时，对应的字符出现在显示窗口中。C（清除）键将擦除到目前为止的所有输入；<（回退）键擦除最后键入的字符。当使用者按下 = 键时，计算表达式的值，并用结果替代显示窗口中的表达式。之后，使用者可以按 C 并键入另一个表达式。如果使用者按下 Q（退出）键，计算器停止工作，并从屏幕上消失。

10.（游戏）你知道如何在迷宫中找到通路吗？写完这个程序后，你永远也不会迷路了！

假定迷宫是方格的矩形数组，有些块被封上了用来表示墙。迷宫有一个入口和一个出口。例如，如果 X 表示墙，则一个迷宫可能是下面这样的：

```
XXXXXXXXXXXXXXXXXX X
X X XXXX X
X XXXXX XXXXX XX X
X XXXXX XXXXXXXX XX X
X X XX XX X
X XXXXXXXXXX XX X
XXXXXXXXXXXXoXXXXXXX
```

一个生物，在前面这个图中用 o 表示，坐在迷宫的入口处（最下面一行）。假定该生物可以朝 4 个方向移动：向北、向南、向东和向西。在图中，向北是往上走，向南是往下走，向东是往右走，向西是往左走。程序让生物从入口穿过迷宫移到出口（最上面一行），如果可行。当生物移动时，它应该标记其路径。在穿过迷宫的旅程结论中，就能看到正确的路径及不正确的尝试了。注意，有些迷宫可能有多条成功路径，而有些可能没有路径。

迷宫中的每个方格都是 4 种状态之一：CLEAR（方格是空的）、WALL（方格是封住的，表示墙的部分）、PATH（方格位于到出口的路径中）和 VISITED（访问过的方格，但位于通向死胡同的

路上)。

这个问题用到两个交互的 ADT。ADT 生物表示生物的当前位置，并包含移动生物时的操作。生物可以向北、南、东和西方向移动，每次移动一格。它应该能报告自己的位置并标记轨迹。ADT 迷宫表示迷宫本身，这是二维排列的方格矩阵。可以从最上面的方格行开始从 0 开始编号，从最左面的方格列开始从 0 开始编号。然后使用行号和列号可唯一表示迷宫中的任一个方格。显然，这个 ADT 需要一个数据结构来表示迷宫。它还需要用方格数给出的表示迷宫高度和宽度的数据，及迷宫入口和出口的行列坐标。

ADT 迷宫还应该包含操作，如使用给定的能详细显示迷宫的描述数据来创建一个具体的迷宫，测试一个方格是否是墙的部分，查看一个方格是否是路径的部分，等等。

表示一个迷宫的输入数据是简单的。例如，前面给出的迷宫可由下列保存在一个文本文件中的输入行表示：

20 7 ←迷宫的宽度和高度的方格数

0 18 ←迷宫出口的行列坐标

6 12 ←迷宫入口的行列坐标

XXXXXXXXXXXXXXXXXX X

X X XXXX X

X XXXX XXXX XX X

X XXXX XXXXXX XX X

X X XX XX X

X XXXXXXXXX XX X

XXXXXXXXXXXX XXXXXX

文件中前 3 行的数值数据之后的每一行都对应于迷宫中的一行，一行中的每个字符对应于迷宫中的一列。X 表示封死的方格(墙的部分)，空格表示空方格。这种记号很方便，因为在你设计迷宫时就能看出迷宫的样子。

使用基于栈的算法找到通过迷宫的路径。查找算法及其支持方法都在 ADT 生物和 ADT 迷宫之外。所以迷宫和生物都是必须传给这些方法的参数。

- 11.(游戏) 考虑第 1 章项目 9 中的洞穴系统。假定你仅可以从一个洞穴进入这个系统，并且从另一个洞穴退出系统。现在设想，你在这个洞穴系统的某个洞穴中醒来。有一个标志表明，从你当前位置看出口在 5 个洞穴中。设计一个算法，查找穿过洞穴的出口。

提示：当你访问每个洞穴时，打标记并把它放到栈中。如果已经访问了与当前洞穴相连的所有洞穴但还没有找到出口，则从栈中弹出这个洞穴。

- 12.(财务) 假设你投资股票市场。每只股票的价值每天可能都在变化，它可能不同于最初的购买成本。股票投资组合的净值是原始成本与其当前价值的差额。设计一个记录你投资的系统。包括购买股票的算法、从投资组合中卖出股票的算法及计算投资组合当前净利的算法。当你卖出股票时，首先卖出最近购买的股票。

提示：你每次购买的股票有  $n$  股，每股  $d$  美元。可以将每次的购买放到一个栈中。如果只卖了部分某只股票，则从栈中删除购买项，修改它以反映剩余的股数，然后再放回栈中。

- 13.(电子商务) 考虑一家维护某种健身手环库存的公司，手环的价格每天都在变化。库存中一个商品的价值是它的当前价格，这可能不同于最初的购买价。健身手环的销售价是其当前价值的 120%。公司最先销售最近购买的手环。当前整个库存的净利是其原始价格与当前销售价格之差。

为这个公司设计一个库存系统。包括购买商品进库的算法、从库存中销售商品的算法，及计算库存当前净利的算法。

提示：公司每次购买  $n$  个手环，每只价格  $d$  美元。可以将每次的购买放到一个栈中。如果只卖了一部分，则从栈中删除购买项，修改它以反映剩余的库存，然后再放回栈中。

# 栈的实现

**先修章节:** 第2章、第3章、第4章、第5章

## 目标

学习完本章后，应该能够

- 使用链表、数组或是向量实现 ADT 栈
- 对不同的实现方式及其性能进行对比分析

本章介绍的 ADT 栈的两种实现方式，用到了之前实现 ADT 包时用过的技术。我们将依次使用结点链表和数组保存栈中的项。本章还将介绍 Java 标准类库中的标准类 `Vector`，使用 `Vector` 实例表示一个栈。你会惊喜地发现这些实现是如此简单和高效。

## 链式实现

**6.1** ADT 栈中的每个操作 `push`、`pop` 和 `peek` 都涉及栈顶。如果使用结点链表来实现栈，那么栈顶元素应该放置在链表的什么位置呢？如果仅有链表的头引用，则添加、删除或是访问第一个结点要快于其他结点。所以，如果链表中第一个结点指向栈顶元素，则栈操作会是最快的，如图 6-1 所示。

还注意到，图中链表的每个结点都指向链表中的一项。仅当新项需要时才分配（即创建）结点。当删除项时它们被释放。回忆第3章段3.24，Java运行时环境自动回收或释放程序不再引用的内存，不需要程序员写语句来释放。

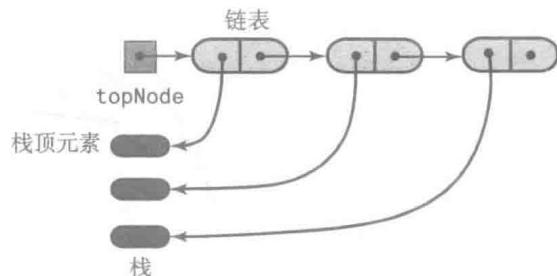


图 6-1 实现栈的结点链表

 **注:** 如果使用结点链表实现栈，则首结点应该指向栈顶元素。

**6.2** **类的框架。** 栈的链式实现有一个数据域 `topNode`，它是结点链表的头引用。默认构造方法将该域的值设置为 `null`。类的框架见程序清单 6-1。

链表中的每个结点都是 `LinkedStack` 类中定义的私有类 `Node` 的实例。这个类有设置和获取方法，很像是在第3章程序清单 3-4 中为 ADT 包定义的那个。

### 程序清单 6-1 链式实现 ADT 栈的类框架

```

1 /**
2 * A class of stacks whose entries are stored in a chain of nodes.
3 */
4 public final class LinkedStack<T> implements StackInterface<T>
5 {
6 private Node topNode; // References the first node in the chain
7

```

```

8 public LinkedStack()
9 {
10 topNode = null;
11 } // end default constructor
12 < Implementations of the stack operations go here. >
13 .
14
15 private class Node
16 {
17
18 private T data; // Entry in stack
19 private Node next; // Link to next node
20
21 < Constructors and the methods getData, setData, getNextNode, and setNextNode
22 are here. >
23 }
24 } // end LinkedStack

```

在栈顶添加。将项入栈的过程是，先分配一个新结点，它指向栈的现有链表，如图 6-2a 所示。6.3 这个引用保存在 `topNode` 中，它是链表的头引用。然后将 `topNode` 指向新结点，如图 6-2b 所示。方法 `push` 的定义如下：

```

public void push(T newEntry)
{
 Node newNode = new Node(newEntry, topNode);
 topNode = newNode;
} // end push

```

可以用下面的语句替换上述方法体中的两条语句：

```
topNode = new Node(newEntry, topNode);
```

这个操作不涉及栈中的其他项。所以性能是  $O(1)$  的。

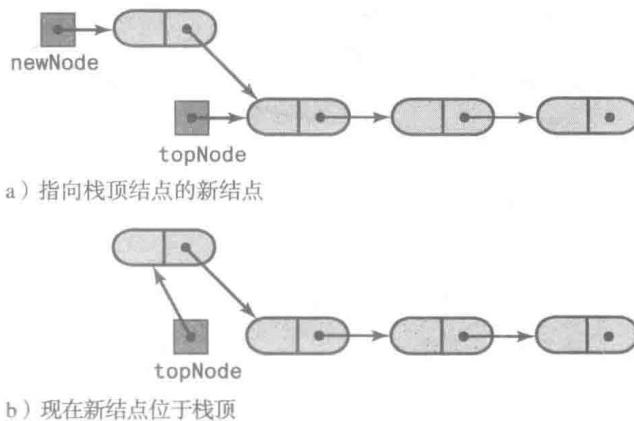


图 6-2 将新结点添加到链式栈的栈顶

取回栈顶。访问链表中首结点的数据部分就可以得到栈顶项。故与 `push` 一样，`peek` 操作也是  $O(1)$  的。注意，如果栈为空，则 `peek` 抛出一个例外。6.4

```

public T peek()
{
 if (isEmpty())
 throw new EmptyStackException();
 else
 return topNode.getData();
} // end peek

```

6.5 删 除 栈 顶。将首结点中的引用赋给 topNode，从而出栈或是删除了栈顶项。故 topNode 将指向链表中的第二个结点，如图 6-3 所示。此外，已没有引用指向原来的首结点，所以它将被释放。因为还想在删除之前返回栈顶项，故方法 pop 的实现如下所示。

```
public T pop()
{
 T top = peek(); // Might throw EmptyStackException
 // Assertion: topNode != null
 topNode = topNode.getNextNode();
 return top;
} // end pop
```

该操作也是  $O(1)$  的。

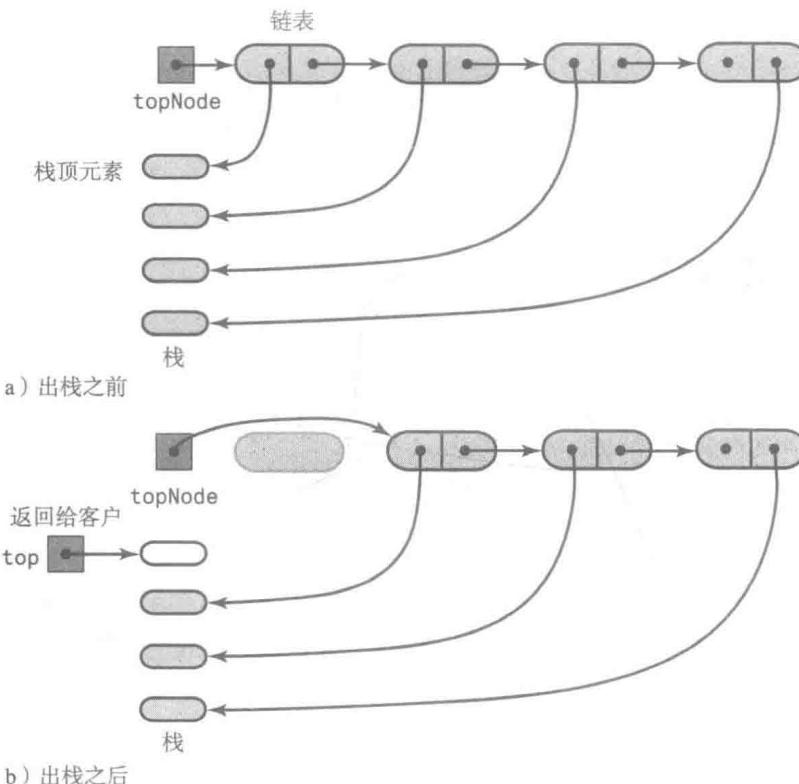


图 6-3 pop 删除链表中首结点前后的栈



#### 安全说明：实现准则

- 使用断言来验证假设。
- 当验证数据时，检查它有效而不是无效。
- 核实给你的返回值，特别是由那些不是你亲自编写的代码提供的返回值。



#### 学习问题 1 修改前面实现的 pop 方法，让其不调用 peek 方法。

类中的其他方法。剩下的公有方法 isEmpty 和 clear 只与 topNode 相关。

```
public boolean isEmpty()
{
```

```

 return topNode == null;
} // end isEmpty

public void clear()
{
 topNode = null;
} // end clear

```

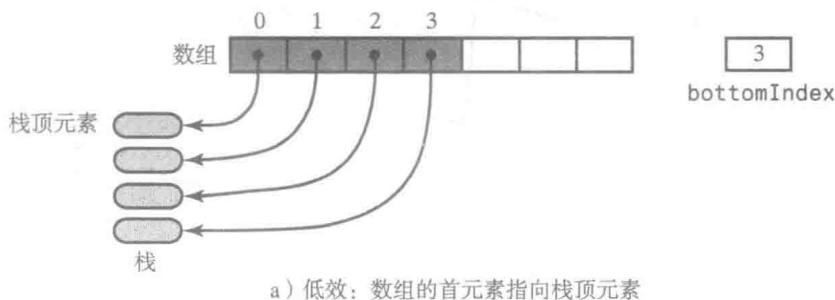


**学习问题 2** 将栈顶位于结点链表的链尾而不是链头，这样实现 ADT 栈合理吗？解释之。

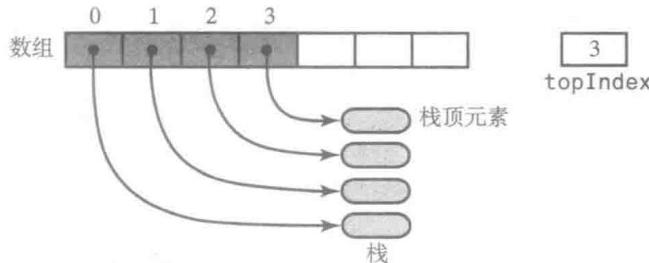
## 基于数组的实现

如果使用数组来实现栈，则栈顶元素应该放在哪儿呢？如果数组的首位置指向栈顶，如图 6-4a 所示，则添加或删除栈元素时都必须移动数组中的所有项。如果数组首位置指向栈底元素，则栈操作的效率更高。这样栈顶元素由数组中已使用的最后一个元素所指，如图 6-4b 所示。这样的布局使得添加或删除栈元素时不需要移动数组中的其他项。所以，通常的基于数组的实现方案中的缺点之一在这里也不复存在。本章最后的练习中考虑了基于数组实现栈的其他实现方案。

注：如果使用数组实现栈，则数组的首元素指向栈底。数组最后的占用元素才指向栈顶元素。



a) 低效：数组的首元素指向栈顶元素



b) 高效：数组的首元素指向栈底元素

图 6-4 栈的两种数组表示

类的框架。基于数组实现的栈中，其数据域有栈元素的数组和栈顶元素的下标。默认构造方法创建一个带默认容量的栈；另一个构造方法让客户指定栈的容量。程序清单 6-2 给出了类框架。

### 程序清单 6-2 基于数组实现 ADT 栈的框架

```

1 /**
2 * A class of stacks whose entries are stored in an array.
3 */
4 public final class ArrayStack<T> implements StackInterface<T>
5 {
6 private T[] stack; // Array of stack entries
7 private int topIndex; // Index of top entry
8 private boolean integrityOK;
9 private static final int DEFAULT_CAPACITY = 50;
10 private static final int MAX_CAPACITY = 10000;
11
12 public ArrayStack()
13 {
14 this(DEFAULT_CAPACITY);
15 } // end default constructor
16
17 public ArrayStack(int initialCapacity)
18 {
19 integrityOK = false;
20 checkCapacity(initialCapacity);
21
22 // The cast is safe because the new array contains null entries
23 @SuppressWarnings("unchecked")
24 T[] tempStack = (T[])new Object[initialCapacity];
25 stack = tempStack;
26 topIndex = -1;
27 integrityOK = true;
28 } // end constructor
29
30 < Implementations of the stack operations go here. >
31 < Implementations of the private methods go here; checkCapacity and checkIntegrity
32 are analogous to those in Chapter 2. >
33 . .
34 } // end ArrayStack

```

为了能表示空栈，让 `topIndex` 的初值为 `-1`。这样处理后，当向数组添加新元素时，`push` 操作中要先让 `topIndex` 的值加 1，然后再使用它的值。

**6.9 在栈顶添加。** `push` 方法调用另外的私有方法 `ensureCapacity`，来检查数组是否还有空间保存新元素。如果有必要，`ensureCapacity` 变长数组，从而避免栈没有空间容纳新项。`push` 方法紧邻数组中最后占用的位置之后放置新元素。

```

public void push(T newEntry)
{
 checkIntegrity();
 ensureCapacity();
 stack[topIndex + 1] = newEntry;
 topIndex++;
} // end push

private void ensureCapacity()
{
 if (topIndex == stack.length - 1) // If array is full, double its size
 {
 int newLength = 2 * stack.length;
 checkCapacity(newLength);
 stack = Arrays.copyOf(stack, newLength);
 } // end if
} // end ensureCapacity

```

注意到，`ensureCapacity` 方法类似于第 2 章见过的 `ResizableArrayBag` 类中的

`ensureCapacity` 方法。这两个私有方法都在数组满后倍增其大小。

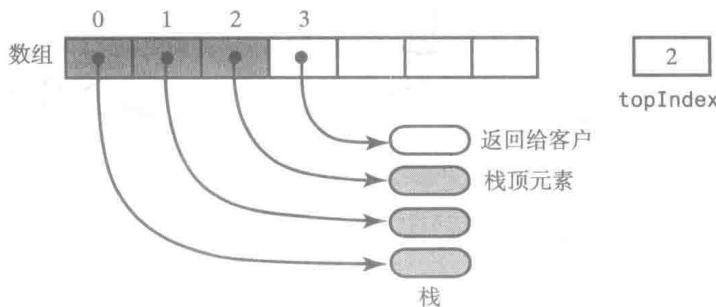
当 `ensureCapacity` 不需要改变数组 `stack` 的大小时, `push` 操作是  $O(1)$  的, 因为其性能与栈的大小无关。但变长数组的操作是  $O(n)$  的, 故当数组满时, `push` 的性能降为  $O(n)$ 。不过出现这种情形后, 下一次的 `push` 又会是  $O(1)$  的。公平起见, 所有 `push` 操作应均摊偶尔为之的变长数组的开销。就是说, 将数组倍增的开销分摊 (amortize) 到栈的所有入栈操作上。除非必须多次变长数组, 否则每次 `push` 几乎都是  $O(1)$  的。

取回栈顶。`peek` 操作或者返回数组中位于 `topIndex` 处的元素, 或者栈空时抛出一个异常。6.10

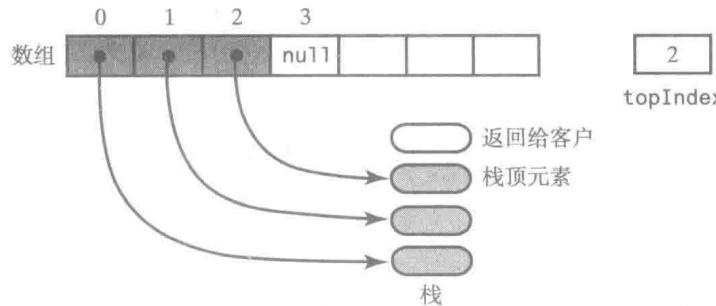
```
public T peek()
{
 checkIntegrity();
 if (isEmpty())
 throw new EmptyStackException();
 else
 return stack[topIndex];
} // end peek
```

这个操作是  $O(1)$  的。

删除栈顶。与 `peek` 一样, `pop` 操作取回栈顶元素, 但随后删除它。要删除图 6-4b 中所示栈的栈顶元素, 只需让 `topIndex` 减 1, 如图 6-5a 所示。这步操作简单但已足够了, 因为其他方法会正确处理。例如, 给定如图 6-5a 所示的栈, `peek` 将返回 `stack[2]` 指向的项。但是, 已经返回给客户的之前的那个栈顶项, 仍由数组元素指向。如果所实现的代码都正确, 这也没什么危害。不过为了安全起见, `pop` 操作中, 在 `topIndex` 值减 1 之前, 可以先将 `stack[topIndex]` 设置为 `null`。图 6-5b 图示了这种情况下的栈。6.11



a) 通过 `topIndex` 减 1 的方式



b) 通过将 `stack[topIndex]` 设置为 `null`, 然后 `topIndex` 值减 1 的方式

图 6-5 基于数组的栈删除栈顶元素的两种不同方式

对应这段说明的 `pop` 实现如下所示。

```

public T pop()
{
 checkIntegrity();
 if (isEmpty())
 throw new EmptyStackException();
 else
 {
 T top = stack[topIndex];
 stack[topIndex] = null;
 topIndex--;
 return top;
 } // end if
} // end pop

```

与 peek 一样，pop 操作是  $O(1)$  的。



### 注：内存使用

和图 6-3 中的链表不同，图 6-4 和图 6-5 都有未用的元素。如果最终用其他的栈项填满了数组，则变长数组，并且会把它更多的未用元素。链表也有自己的不利因素，它使用额外的内存用于结点的链接部分。



**学习问题 3** 修改上面实现的方法 pop，让其调用 peek。

**学习问题 4** 如果要实现基本类型的栈而不是对象的栈，要如何修改 pop 方法？

6.12

isEmpty 和 clear 方法。isEmpty 方法仅用到了 topIndex。

```

public boolean isEmpty()
{
 return topIndex < 0;
} // end isEmpty

```

因为尽管栈是空栈，栈的方法应该仍能正确运行，故 clear 可以简单地将 topIndex 设置为 -1。不过，栈中的对象仍占据着空间。正如 pop 方法中将 stack[topIndex] 设置为 null 一样，clear 也应该将数组中使用过的每个位置设置为 null。另外，clear 可以重复调用 pop，直到栈空时为止。将 clear 的实现留作练习。



**学习问题 5** 如果 stack 是保存栈中元素的数组，那么，将栈顶元素放在 stack[0] 的不足之处是什么？

**学习问题 6** 保存栈元素时，如果先使用数组 stack 的末尾位置，后使用数组的首位置，那么，放在 stack[stack.length-1] 中的应该是栈顶元素还是栈底元素？为什么？

**学习问题 7** 实现方法 clear，将栈中用过的每个数组位置置为 null。

**学习问题 8** 实现方法 clear，重复调用 pop，直到栈空时为止。

## 基于向量的实现

6.13

让栈按需增大的一种办法是将元素保存在变长数组中，如我们实现 ArrayStack 类时的处理一样。另一种办法是使用向量（vector）替代数组。向量是一个对象，其行为类似于高级数组。向量中的项与数组中的项一样也有下标，且从 0 开始。与数组不同的是，向量有设置或是访问项的方法。你可以创建一个指定大小的向量，而且它的大小将按需增长。过程细

节对客户隐藏。

如果将栈的元素保存在一个向量中，则可以使用向量的方法来维护栈中的项。图 6-6 显示的是一个客户通过 `StackInterface` 中的方法与栈进行交互的情景。这些方法的实现反过来又影响向量的方法，从而得到预期的栈处理结果。

向量是标准类 `Vector` 的实例，我们稍后讨论。

## Java 类库：类 `Vector`

Java 类库中包含了类 `Vector`，其实例（称为向量）的行为类似于一个变长数组。下面是实现 ADT 栈时会用到的 `Vector` 的构造方法和方法：

`public Vector()`

创建一个空向量，或类似于数组的容器，初始大小为 10。当向量需要扩展容量时，容量倍增。

`public Vector(int initialCapacity)`

创建一个带指定初始容量的空向量。当向量需要扩展容量时，容量倍增。

`public boolean add(T newEntry)`

将新项添加到向量的末尾。

`public T remove(int index)`

删除向量中指定下标的项并返回。

`public void clear()`

删除向量中的所有项。

`public T lastElement()`

返回向量中的最后一项。

`public boolean isEmpty()`

如果向量为空则返回真。

`public int size()`

返回向量中当前的元素个数。

你可以参看 Java 类库中的在线文档，了解关于 `Vector` 类的更多细节。



### 注：Java 类库：`Vector` 类

Java 类库中的 `Vector` 类在 `java.util` 包中。向量类似于变长数组，其中的元素有从 0 开始的下标。可以使用类中的方法来处理向量。

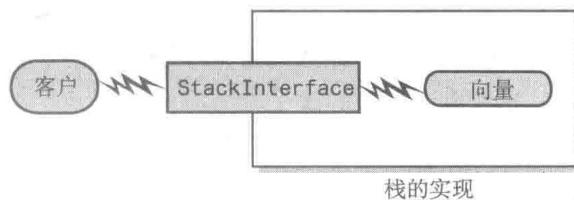


图 6-6 客户使用 `StackInterface` 中所给的方法；这些方法配合向量的方法共同完成栈的操作

6.14

## 使用 Vector 实现 ADT 栈

6.15

使用向量保存栈的元素，类似于使用数组来保存，但会更容易些。我们让向量的第一个元素指向栈底元素。所以，向量看起来如同图 6-4b 中的数组一样。我们不需要维护栈顶元素的下标，但我们能从向量大小推出这个下标，而向量大小很容易得到。另外，向量按需扩展，我们也不用担心这个细节问题。

因为 `Vector` 的实现基于动态可变大小的数组，所以这样实现的栈的性能与上一节给出的基于数组的实现方式是一样的。

 **注：**如果使用向量实现栈，则向量的首元素应该指向栈底项。而向量最后的占用位置的元素则指向栈顶项。

6.16

**类的框架。**实现栈的类首先要声明一个向量作为数据域，并在构造方法中分配向量。故在类定义之前必须提供 `import` 语句。程序清单 6-3 给出了类框架。

**程序清单 6-3** 基于向量实现 ADT 栈的类框架

```

1 /**
2 * A class of stacks whose entries are stored in a vector.
3 */
4 public final class VectorStack<T> implements StackInterface<T>
5 {
6 private Vector<T> stack; // Last element is the top entry in stack
7 private boolean integrityOK;
8 private static final int DEFAULT_CAPACITY = 50;
9 private static final int MAX_CAPACITY = 10000;
10
11 public VectorStack()
12 {
13 this(DEFAULT_CAPACITY);
14 } // end default constructor
15
16 public VectorStack(int initialCapacity)
17 {
18 integrityOK = false;
19 checkCapacity(initialCapacity);
20 stack = new Vector<T>(initialCapacity); // Size doubles as needed
21 integrityOK = true;
22 } // end constructor
23
24 < Implementations of checkIntegrity, checkCapacity, and the stack operations go here. >
25
26 } // end VectorStack

```

6.17

**在栈顶添加。**使用 `Vector` 类的方法 `add`，可以将一个项添加到向量尾，即栈顶。

```

public void push(T newEntry)
{
 checkIntegrity();
 stack.add(newEntry);
} // end push

```

6.18

**取回栈顶。**使用 `Vector` 类的方法 `lastElement`，可以取回栈顶项。

```

public T peek()
{
 checkIntegrity();
 if (isEmpty())

```

```

 throw new EmptyStackException();
 else
 return stack.lastElement();
} // end peek

```

删除栈顶。使用 Vector 类的方法 remove，可以删除栈顶项。这个方法的参数是向量最后一项的下标，因为那个项是栈顶。下标值是向量当前大小 stack.size() 减 1。6.19

```

public T pop()
{
 checkIntegrity();
 if (isEmpty())
 throw new EmptyStackException();
 else
 return stack.remove(stack.size() - 1);
} // end pop

```

类的其他方法。其他的公有方法 isEmpty 和 clear 调用 Vector 类中的类似方法。6.20

```

public boolean isEmpty()
{
 checkIntegrity();
 return stack.isEmpty();
} // end isEmpty

public void clear()
{
 checkIntegrity();
 stack.clear();
} // end clear

```



**学习问题 9** 如果用向量保存栈中的项，则将栈顶元素放在向量的首位置中合理吗？



**注：**因为 Java 实现类 Vector 时使用的是数组，故 VectorStack 是基于数组实现的 ADT 栈。它使用一个变长数组来保存栈的项，所以栈可以按需增长。



**注：**写 VectorStack 肯定要比写本章介绍的基于数组实现的代码容易些。因为 VectorStack 的方法调用了 Vector 的方法，而运行时间要多于 ArrayStack 方法的运行时间。但是，多出的这些时间常常微不足道。

## 本章小结

- 可以使用仅有头引用的结点链表来实现栈。如果链表的首结点指向栈顶项，则栈的操作最快。这是因为添加、删除或是访问链表的首结点要快于对其他结点的操作。
- 链式实现中栈操作都是  $O(1)$  的。
- 可以使用数组实现栈。如果数组的首位置保存栈底元素，则在栈中执行添加或删除操作时不需移动数组元素。
- 变长数组可避免栈满时不能容纳新元素。不过数组中一般会含有未用的位置。
- 基于数组的实现中，栈操作是  $O(1)$  的。但当数组满时，push 要倍增数组大小。这种情况下，push 是  $O(n)$  的。如果将这个额外开销分摊在所有其他的入栈操作上，并且数组倍增也不是很频繁，那么 push 几乎是  $O(1)$  的。
- 可以使用向量实现栈。将栈底元素放在向量的开始位置。

- 因为 `Vector` 的实现是基于动态可变大小的数组的，故基于向量实现的栈的性能类似于基于数组实现的性能。

## 练习

1. 比较基于数组实现 ADT 栈和链式实现 ADT 栈的优缺点。
2. 考虑第 1~3 章中介绍的 ADT 包。
  - a. 能使用包来保存栈中的项从而实现 ADT 栈吗？解释你的答案。
  - b. 能使用栈来保存包中的项从而实现 ADT 包吗？解释你的答案。
3. 假定 ADT 栈包含了方法 `display`，它显示栈中的项，返回值为 `void`。对下面的每个类，实现这个方法。
  - a. 程序清单 6-1 中的 `LinkedStack`。
  - b. 程序清单 6-2 中的 `ArrayList`。
  - c. 程序清单 6-3 中的 `VectorStack`。
  - d. `LinkedStack`、`ArrayList` 或是 `VectorStack` 的任意客户。
4. 定义方法 `toArray` 取代方法 `display`，重做前一个练习。
5. 假定 ADT 栈包含了方法 `remove(n)`，它从栈中删除最上面的  $n$  个元素，返回值为 `void`。通过注释和方法头来规范说明该方法。考虑对包含的元素个数不足  $n$  个的栈的可能处理结果。
6. 定义上题中描述的 `remove(n)`，取代方法 `display`，重做练习 3。
7. 在 ADT 栈的链式实现中，若将栈顶项放在链表的尾结点处。详述如何定义栈的操作 `push`、`pop` 和 `peek`，以避免遍历链表。
8. 段 6.9 说明，基于数组栈的 `push` 方法通常是  $O(1)$  的，但当栈需要倍增其大小时，`push` 是  $O(n)$  的。不过，这个结论并不像表面上那么差。假定你将栈的大小从  $n$  个元素倍增到  $2n$  个元素。
  - a. 在栈再次倍增前能调用多少次 `push`？
  - b. 这些 `push` 调用每次都是  $O(1)$  的，所有 `push` 操作的平均花费是多少？（平均花费是指所有 `push` 调用的总花费除以 `push` 调用的次数。）
9. 假定在基于数组实现的栈中，当栈满时不是倍增它的大小，而是将数组增加到由某个正常数  $k$  指定的大小。
  - a. 如果有一个空栈，使用初始时大小为  $k$  的数组，执行  $n$  次入栈操作，则将执行多少次增大操作？假定  $n > k$ 。
    - b.  $n$  次入栈操作的平均花费是多少？
10. 假定在基于数组实现的栈中，当栈满时不是倍增其大小，而是对某个正常数  $k$ ，让数组按序列  $3k$ 、 $5k$ 、 $7k$ 、 $9k$ …增大。
  - a. 如果有一个空栈，使用初始时大小为  $k$  的数组，执行  $n$  次入栈操作，则将执行多少次增大操作？假定  $n > k$ 。
    - b.  $n$  次入栈操作的平均花费是多少？
11. 当数组满时，可以倍增其大小或是使用练习 9 和练习 10 中所描述的方案。这 3 种方案各自的优缺点是什么？
12. 若在基于数组实现的栈上有若干栈操作。假定数组倍增其大小，但后来，只有不到一半的数组位置被栈实际占用。描述此种情形下数组大小减半的实现。这种实现方式的优缺点是什么？

## 项目

1. 使用数组 `stack` 保存栈中元素来实现 ADT 栈。需要时动态扩展数组。栈底元素放在 `stack[stack.length - 1]` 中。

2. 将栈顶元素放在 `stack[stack.length - 1]` 中，重做项目 1。
3. 将栈顶元素放在 `stack[0]` 中，重做项目 1。
4. 编写代码，实现练习 7 中描述的 ADT 栈。
5. ADT 栈可收回栈顶元素但不删除它。对栈的某些应用，还需要在不删除栈顶元素的前提下，收回栈顶元素的下一个元素。称这样的操作为 `peek2`。如果栈中元素个数多于 1 个，则 `peek2` 返回从栈顶起第二个元素且不改变栈。如果栈中元素少于 2 个，则 `peek2` 抛出一个异常。实现包含 `peek2` 方法的链式栈。
6. 当客户试图从空栈中收回或删除一项时，栈抛出一个异常。另一种方式是返回 `null`。
  - a. 修改接口 `StackInterface`，在这些情形下返回 `null`。
  - b. 修改基于数组实现的栈，以符合你对 `StackInterface` 的修改。编写程序验证这些修改。
  - c. 使用链式栈重做 b。
7. 假定你希望在倍增机制外，还实现练习 9 和练习 10 中描述的重定大小的机制。
  - a. 编写新的基于数组实现的栈，让客户在创建栈时规范说明数组如何变长的机制及相关的常数。
  - b. 编写程序验证这些修改。
  - c. 添加了在栈创建后允许客户改变数组变长机制和相关常数的方法，讨论其优缺点。
8. 实现 ADT 包，使用向量保存栈元素。
9. 修改第 2 章的 `ArrayBag` 和第 3 章的 `LinkedBag`，让它们实现接口 `BagInterface`，如第 5 章练习 14 中的修改。
10. (游戏) 实现第 5 章项目 11 中描述的洞穴逃离算法。
11. (财务) 实现第 5 章项目 12 中描述的股票投资系统。
12. (电子商务) 实现第 5 章项目 13 中描述的库存系统。

# 再论异常

先修章节：附录 C、Java 插曲 2

本插曲中，继续学习 Java 插曲 2 开始的异常。

## 程序员定义的异常类

你可以从已有的异常类派生，来定义自己的异常类。已有的超类可能是 Java 类库中的一个，或是你自己的。异常子类中的构造方法是你必须定义的最重要的——且常常是唯一的——方法。其他的方法都从超类中继承。

J3.1

 定义示例。例如，考虑 Java 的 Math 类中定义的计算实数平方根的方法 sqrt。因为 sqrt 返回一个 double 类型的值，所以它仅计算非负数的平方根。如果给方法一个负数，它会返回特殊值 NaN，这表示“不是一个数”。如果显示的话，这个值将显示为 NaN。如果用在算术运算中，则结果是 NaN。

设想另一种情况，当把一个负数传给这个方法时，平方根方法不是返回 NaN，而是抛出一个运行时异常。方法肯定能抛出一个 RuntimeException 实例，但抛出一个更具体的异常会更好些。所以我们定义自己的类 SquareRootException，列在程序清单 JI3-1 中。因为想要一个运行时异常，故我们的类派生于 RuntimeException。这个类的两个构造方法中的每一个都使用 super 来调用 RuntimeException 的构造方法，将一条信息作为字符串传递给它。默认的构造方法传递一条默认信息，而第二个构造方法传递的是其被调用时实参提供的信息。程序员定义的大多数异常类都与 SquareRootException 一样简单。

### 程序清单 JI3-1 异常类 SquareRootException

```

1 /**
2 * A class of runtime exceptions thrown when an attempt
3 * is made to find the square root of a negative number.
4 */
5 public class SquareRootException extends RuntimeException
6 {
7 public SquareRootException()
8 {
9 super("Attempted square root of a negative number.");
10 } // end default constructor
11
12 public SquareRootException(String message)
13 {
14 super(message);
15 } // end constructor
16 } // end SquareRootException

```

J3.2

注意，SquareRootException 的默认构造方法可以使用 this 来替代 super，如下所示：

```
public SquareRootException()
{
 this("Attempted square root of a negative number.");
} // end default constructor
```

通过使用 `this`, 这个构造方法调用我们新类中的第二个构造方法, 后者再调用 `RuntimeException` 类的构造方法。与之相反, 列在程序清单 JI3-1 中的默认构造方法使用 `super` 来直接调用 `RuntimeException` 的构造方法。虽然程序清单 JI3-1 中的版本更直接, 也似乎更好, 但使用 `this` 将一个构造方法与同类中的另一个关联起来, 通常更可取, 因为重复的代码更少些, 故这样做会减少错误。

 使用自己的异常类。我们已经假定, 平方根方法当所给实参为负数时会抛出一个运行时异常。现在, 我们已有一个合适的异常类, 现在来定义静态方法类内的这个方法, 列在程序清单 JI3-2 中, 这个静态方法类非常类似于 `Math` 类。JI3.3

方法 `squareRoot` 的头部类似于 `Math.sqrt` 的头部, 不过还包括了 `throws` 子句, 表示该方法可能会抛出一个 `SquareRootException` 异常。要注意方法头部之前的 javadoc 注释中的 `@throws` 标签。回忆 Java 插曲 2 中提到, 这个标签标识的是对可能发生的异常的描述。因为 `SquareRootException` 是运行时异常, 所以可以将它列在 `throws` 子句中, 且可在 javadoc 注释中进行说明。

在 `squareRoot` 的方法体内有一个 `throw` 语句。如果方法的实参是负数, 则方法抛出 `SquareRootException`。如果实参不是负的, 那么方法只返回由 `Math.sqrt` 方法计算得到的平方根。

### 程序清单 JI3-2 类 OurMath 和其静态方法 squareRoot

```
1 /**
2 * A class of static methods to perform various mathematical
3 * computations, including the square root.
4 */
5 public class OurMath
6 {
7 /** Computes the square root of a nonnegative real number.
8 * @param value A real value whose square root is desired.
9 * @return The square root of the given value.
10 * @throws SquareRootException if value < 0. */
11 public static double squareRoot(double value) throws SquareRootException
12 {
13 if (value < 0)
14 throw new SquareRootException();
15 else
16 return Math.sqrt(value);
17 } // end squareRoot
18
19 < Other methods not relevant to this discussion are here. >
20
21 } // end OurMath
```

类 `OurMath` 的论证示例列在程序清单 JI3-3 中。注意作为异常的结果而显示的信息。还要注意到, 当异常发生时停止执行。JI3.4

### 程序清单 JI3-3 类 OurMath 的驱动程序

```

1 /**
2 * A demonstration of a runtime exception using the class OurMath.
3 */
4 public class OurMathDriver
5 {
6 public static void main(String[] args)
7 {
8 System.out.print("The square root of 9 is ");
9 System.out.println(OurMath.squareRoot(9.0));
10
11 System.out.print("The square root of -9 is ");
12 System.out.println(OurMath.squareRoot(-9.0));
13
14 System.out.print("The square root of 16 is ");
15 System.out.println(OurMath.squareRoot(16.0));
16 } // end main
17 } // end OurMathDriver

```

#### 输出

```

The square root of 9 is 3.0
The square root of -9 is Exception in thread "main" SquareRootException:
Attempted square root of a negative number.
 at OurMath.squareRoot(OurMath.java:14)
 at OurMathDriver.main(OurMathDriver.java:12)

```

J3.5



假定类 OurMath 被其他程序员广泛使用。Joe 想使用这个类来计算平方根，但当 squareRoot 遇到负数实参时他不想接收错误信息。而是想让方法返回一个用 i 表示的复数<sup>⊖</sup>，i 是  $-1$  的平方根的缩写。例如， $-9$  的平方根是  $3i$ ，因为

$$\sqrt{-9} = \sqrt{9(-1)} = \sqrt{9}\sqrt{-1} = 3i$$

为了适应包括 i 及不含 i 的结果，Joe 有自己的返回一个字符串的方法。所以 Joe 设想，方法应该返回字符串 "3i" 作为  $-9$  的平方根，而 9 的平方根应该返回字符串 "3"。

Joe 的方法会调用 OurMath.squareRoot。因为这个调用必须出现在 try 块中，所以 Joe 写了如下的语句：

```

String result = "";
try
{
 Double temp = OurMath.squareRoot(value);
 result = temp.toString();
}

```

只要 value 不是负数一切都好，但如果它是负数，则抛出 SquareRootException。Joe 不想像程序清单 JI3-3 所示的驱动程序那样显示一条错误信息，而是，想让他的方法对负数实参也返回正确的值。他在纸上用下列伪代码写出了他的想法：

```

//假定 value 是负数
catch (SquareRootException e)
{
 Double temp = -value 的平方根
 result = temp.toString() 加上 "i"
}

```

<sup>⊖</sup> 这里仅需要了解复数的基本知识。复数在实数基础上增加了含有 i 的虚部。每个复数都有  $a+bi$  的形式，其中 a 和 b 都是实数。要使用这种形式表示实数，可让 b 为 0。

然后 Joe 将这个伪代码转换成下面的 catch 块：

```
catch (SquareRootException e)
{ // Assertion: value is negative
 Double temp = OurMath.squareRoot(-value);
 result = temp.toString() + "i";
}
```

程序清单 JI3-4 显示了 Joe 的类 JoeMath 中的方法 squareRoot，程序清单 JI3-5 则给出了这个类的论证示例。J3.6

#### 程序清单 JI3-4 类 JoeMath

```
1 /**
2 A class of static methods to perform various mathematical
3 computations, including the square root.
4 */
5 public class JoeMath
6 {
7 /** Computes the square root of a real number.
8 @param value A real value whose square root is desired.
9 @return A string containing the square root. */
10 public static String squareRoot(double value)
11 {
12 String result = "";
13 try
14 {
15 Double temp = OurMath.squareRoot(value);
16 result = temp.toString();
17 }
18 catch (SquareRootException e)
19 {
20 Double temp = OurMath.squareRoot(-value);
21 result = temp.toString() + "i";
22 }
23
24 return result;
25 } // end squareRoot
26
27 < Other methods not relevant to this discussion could be here. >
28
29 } // end JoeMath
```

#### 程序清单 JI3-5 类 JoeMath 的驱动程序

```
1 /**
2 A demonstration of a runtime exception using the class JoeMath.
3 */
4 public class JoeMathDriver
5 {
6 public static void main(String[] args)
7 {
8 System.out.print("The square root of 9 is ");
9 System.out.println(JoeMath.squareRoot(9.0));
10
11 System.out.print("The square root of -9 is ");
12 System.out.println(JoeMath.squareRoot(-9.0));
13
14 System.out.print("The square root of 16 is ");
15 System.out.println(JoeMath.squareRoot(16.0));
16
17 System.out.print("The square root of -16 is ");
```

```

18 System.out.println(JoeMath.squareRoot(-16.0));
19 } // end main
20 } // end JoeMathDriver

```

**输出**

```

The square root of 9 is 3.0
The square root of -9 is 3.0i
The square root of 16 is 4.0
The square root of -16 is 4.0i

```

## 继承和异常

J3.7 设想一个类，其方法 `someMethod` 的头部有一个 `throws` 子句。如果我们在它的子类中重写了 `someMethod`，那么能在它的 `throws` 子句中列出其他的受检异常吗？不能，Java 不会让我们这样做的；如果这样做会得到一条语法错误的信息。

例如，考虑下列超类和子类：

```

public class SuperClass
{
 public void someMethod() throws Exception1
 {
 . .
 } // end someMethod
} // end SuperClass

public class SubClass extends SuperClass
{
 public void someMethod() throws Exception1, Exception2 // ERROR!
 {
 . .
 } // end someMethod
} // end SubClass

```

重写方法中的 `throws` 子句将标记为语法错误。现在来考虑为什么这是一个错误。

假定程序创建了 `SubClass` 的一个实例，将这个对象赋给 `SuperClass` 的一个变量——称之为 `superObject`——并将调用 `superObject.someMethod()` 放在 `try` 块内，如下所示：

```

public class Driver
{
 public static void main(String[] args)
 {
 SuperClass superObject = new SubClass();
 try
 {
 superObject.someMethod();
 }
 catch (Exception1 e)
 {
 System.out.println(e.getMessage());
 }
 } // end main
} // end Driver

```

因为 `superObject` 指向 `SubClass` 的实例，所以调用的是 `someMethod` 在 `SubClass` 中的版本。但因为 `superObject` 的静态类型是 `SuperClass`，故编译程序仅查看 `someMethod` 在 `SuperClass` 中的定义。所以，它只会检查 `Exception1` 被捕获的情况。如果 `SubClass` 中的 `throws` 子句是合法的，则可能会调用 `SubClass` 中的 `someMethod` 方法但不去捕获 `Exception2` 异常。

J3.8

如果异常也使用了继承，则控制哪些异常可以出现在重写方法的 `throws` 子句中的规则

相对较灵活。例如，如果 `Exception2` 继承于 `Exception1`，则下列代码是合法的：

```
public class SuperClass
{
 public void someMethod() throws Exception1
 {
 . .
 } // end someMethod
} // end SuperClass

public class SubClass extends SuperClass
{
 public void someMethod() throws Exception2 // OK, assuming Exception2
 {
 . .
 } // end someMethod
} // end SubClass
```

 **注：**在超类中被重写方法的 `throws` 子句中没有列出的异常，也不能列在子类中重写方法的 `throws` 子句中，除非它们是从被重写方法列出的异常类所派生来的。不过，重写方法可以在其 `throws` 子句中少列或完全不列异常。

## finally 块

如果你有不论异常是否发生都必须要执行的代码，则可以将它放到 `try` 块的最后及每个 `catch` 块的最后。不过，可以有一个简单的办法来完成这件事，将所说的这段代码放到 `finally` 块中，并放在最后一个 `catch` 块的后面。`finally` 块内的代码在 `try` 块或一个执行的 `catch` 块结束后执行。虽然 `finally` 块可有可无，但这个块是提供清理服务的一个好办法，例如关闭一个文件或是释放系统资源。

J3.9

下列代码显示 `finally` 块的位置：

```
try
{
 < 可能抛出异常的代码，或是执行了一条 throw 语句，或是调用了一个抛出异常的方法 >
}
catch (AnException e)
{
 < 处理 AnException 类或是 AnException 的子类的异常的代码 >
}
< 可能的处理其他类型异常的 catch 块 >
finally
{
 < try 块或是执行 catch 块结束后执行的代码 >
}
```

 **注：**不论异常是否发生，都会执行 `finally` 块内的语句，但如果 `try` 块或一个 `catch` 块内调用了 `System.exit`，则这些语句不会被执行。如果没有发生异常，则 `finally` 块在其对应的 `try` 块执行完后执行。（如果 `try` 块含有一个 `return` 语句，则 `finally` 块在 `return` 之前执行。）不过，如果发生异常，则它由一个 `catch` 块捕获，执行完 `catch` 块后再执行 `finally` 块。

 **示例。**假定你打开冰箱门找牛奶。不管找到没有，都应该关上门。下列代码中，如果没有找到牛奶，则方法 `takeOutMilk` 将抛出一个异常。不管异常发生与否，都在 `finally` 块内调用 `closeRefrigerator`。

J3.10

```

try
{
 openRefrigerator();
 takeOutMilk();
 pourMilk();
 putBackMilk();
}
catch (NoMilkException e)
{
 System.out.println(e.getMessage());
}
finally
{
 closeRefrigerator();
}

```

J3.11

现在执行前面这个示例中给出的代码，来明确说明 `finally` 块的行为。本示例中要调用的方法的简单定义列在程序清单 JI3-6 中。除 `takeOutMilk` 方法外其他方法都只显示一条相应的信息。而方法 `takeOutMilk` 在某个随机时间显示一条信息，在其他时间抛出 `NoMilkException`。

在程序清单 JI3-6 所示的第一个示例输出中，没有异常发生。`try` 块内的每个方法依次执行，如输出所示。最后，执行 `finally` 块内的方法 `closeRefrigerator`。在第二个示例输出中，正常执行 `openRefrigerator`，之后 `takeOutMilk` 抛出一个异常。`catch` 块捕获异常后，如期执行 `finally` 块。

### 程序清单 JI3-6 finally 块的论证示例

```

1 /**
2 * Demonstrates the behavior of a finally block.
3 */
4 public class GetMilk
5 {
6 public static void main(String[] args)
7 {
8 try
9 {
10 openRefrigerator();
11 takeOutMilk();
12 pourMilk();
13 putBackMilk();
14 }
15 catch (NoMilkException e)
16 {
17 System.out.println(e.getMessage());
18 }
19 finally
20 {
21 closeRefrigerator();
22 }
23 } // end main
24
25 public static void openRefrigerator()
26 {
27 System.out.println("Open the refrigerator door.");
28 } // end openRefrigerator
29
30 public static void takeOutMilk() throws NoMilkException
31 {
32 if (Math.random() < 0.5)

```

```
33 System.out.println("Take out the milk. ");
34 else
35 throw new NoMilkException("Out of Milk!");
36 } // end openRefrigerator
37
38 < The methods pourMilk, putBackMilk, and closeRefrigerator are analogous to
39 openRefrigerator and are here. >
40
41 } // end GetMilk
```

#### 示例输出1（没有抛出异常）

```
Open the refrigerator door.
Take out the milk.
Pour the milk.
Put the milk back.
Close the refrigerator door.
```

#### 示例输出2（抛出异常）

```
Open the refrigerator door.
Out of milk!
Close the refrigerator door.
```

# 队列、双端队列和优先队列

先修章节：序言、第 5 章

## 目标

学习完本章后，应该能够

- 描述 ADT 队列的操作
- 使用队列来模拟排队
- 程序中使用队列按先进先出方式组织数据
- 描述 ADT 双端队列的操作
- 程序中使用双端队列按时间顺序组织数据，且能对最老和最新的项进行操作
- 描述 ADT 优先队列的操作
- 程序中使用优先队列根据优先级组织数据

排队等待是生活中的一个现象。大多数人都曾花时间在商店、银行或是电影院前排过队。你或许在等待航空公司的代表或技术支持人员的电话，或许等待你要打印的文件传到计算机实验室的打印机上。每个例子中，人们都希望他们能在比他们晚到的人之前得到服务。即先来先服务。

队列是排队的另一个名字，这是我们要在本章研究的一种 ADT 的名字。队列用在操作系统中，且用来模拟现实世界中的事件——即进程或是事件必须等待时会用到队列。

有时你需要比队列允许的更多的灵活性。双端队列像队列一样组织数据，但允许你对最老和最新的项进行操作。当对象的重要性依赖于规则而不是到达时间时，你可以给它指定一个优先级。可以将这样的对象按优先级而不是按时间顺序组织在优先队列中。

队列、双端队列和优先队列是本章要讨论的三种 ADT。

## ADT 队列

与栈一样，ADT 队列（queue）将项按其添加的顺序组织。栈有后进先出，或叫 LIFO 的行为，而队列有先进先出（First-In, First-Out），或叫 FIFO 的行为。要做到这一点，队列的所有添加都在它的后端（back，队尾）进行。这样最近添加的项在队列的后端。最早添加的项在队列的前端（front，队头）。图 7-1 是常见的队列示例。



注：队列的项之间，第一个添加的项，或最早的项，在队列的队头，最后添加的项在队列的队尾。

队列与栈一样，限制对其中的项的访问。虽然有人能插队，但软件队列中的添加必须在队尾进行。客户仅能看到或删除队头的项。能看到不在队头的项的唯一办法是，重复地从队列中删除项，直到所需的项到达队头。如果你一个个地删除队列中的所有项，能得到按时间顺序排列的项，最前面的是第一个添加到队列中的项。

队列没有查找操作。项的值与队列无关，也与项在队列中的位置无关。

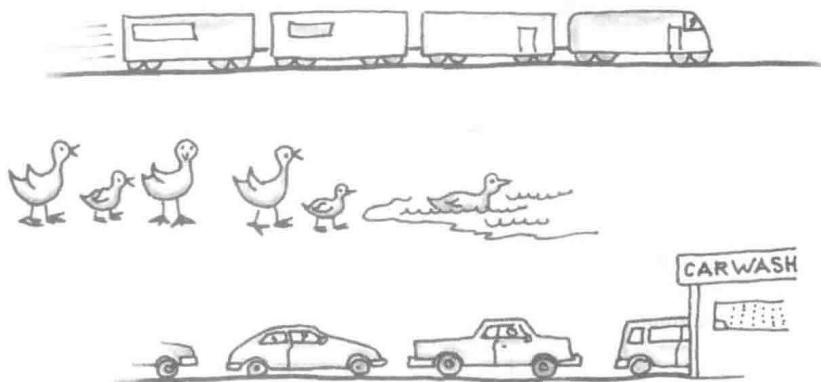


图 7-1 一些日常的队列

向队列中添加项的操作，传统上称为入队（enqueue，念“N-Q”）。删除一项的操作是出队（dequeue，念“D-Q”）。获取队头项的操作称为取值（getFront）。下列规范说明定义了 ADT 队列的一组操作：

| 抽象数据类型：队列         |                                   |                                                 |
|-------------------|-----------------------------------|-------------------------------------------------|
| 数据                |                                   |                                                 |
| 操作                |                                   |                                                 |
| 伪代码               | UML                               | 描述                                              |
| enqueue(newEntry) | +enqueue(newEntry: integer): void | 任务：将新项添加到队尾<br>输入：newEntry 是新项<br>输出：无          |
| dequeue()         | +dequeue(): T                     | 任务：删除并返回队头项<br>输入：无<br>输出：返回队头项。操作之前如果队列为空则抛出异常 |
| getFront()        | +getFront(): T                    | 任务：获取队头项且不改变队列<br>输入：无<br>输出：返回队头项。如果队列为空，则抛出异常 |
| isEmpty()         | +isEmpty(): boolean               | 任务：检查队列是否为空<br>输入：无<br>输出：如果队列为空则返回真            |
| clear()           | +clear(): void                    | 任务：从队列中删除所有的项<br>输入：无<br>输出：无                   |



### 注：方法的另外的名字

正如第 5 章提到过的，类的设计者常常为某些方法定义了别名。对于队列，你可以包含另外的方法 put 和 get，表示入队和出队。名字 add、insert、remove 和 delete 也是合理的别名。同样，你可以提供方法 peek 表示 getFront。

程序清单 7-1 中的 Java 接口规范说明了对象的队列。泛型 T——可以是任何类类型——表示队列中项的数据类型。注意，我们必须定义 EmptyQueueException。将这个定义作为运行时异常留作练习。

**程序清单 7-1** ADT 队列的接口

```

1 public interface QueueInterface<T>
2 {
3 /** Adds a new entry to the back of this queue.
4 @param newEntry An object to be added. */
5 public void enqueue(T newEntry);
6
7 /** Removes and returns the entry at the front of this queue.
8 @return The object at the front of the queue.
9 @throws EmptyQueueException if the queue is empty before the operation. */
10 public T dequeue();
11
12 /** Retrieves the entry at the front of this queue.
13 @return The object at the front of the queue.
14 @throws EmptyQueueException if the queue is empty. */
15 public T getFront();
16
17 /** Detects whether this queue is empty.
18 @return True if the queue is empty, or false otherwise. */
19 public boolean isEmpty();
20
21 /** Removes all entries from this queue. */
22 public void clear();
23 } // end QueueInterface

```

7.4



**示例：展示队列方法。**下列语句在队列中添加、获取及删除字符串。假定类 `LinkedQueue` 实现了 `QueueInterface`，且是可用的。

```

QueueInterface<String> myQueue = new LinkedQueue<>();
myQueue.enqueue("Jada");
myQueue.enqueue("Jess");
myQueue.enqueue("Jazmin");
myQueue.enqueue("Jorge");
myQueue.enqueue("Jamal");

String front = myQueue.getFront(); // Returns "Jada"
System.out.println(front + " is at the front of the queue.");
front = myQueue.dequeue(); // Removes and returns "Jada"
System.out.println(front + " is removed from the queue.");
myQueue.enqueue("Jerry"); // Adds "Jerry"
front = myQueue.getFront(); // Returns "Jess"
System.out.println(front + " is at the front of the queue.");
front = myQueue.dequeue(); // Removes and returns "Jess"
System.out.println(front + " is removed from the queue.");

```

图 7-2a ~ 图 7-2e，说明的是队列的头 5 次添加。添加之后，队列——从队头至队尾——含有字符串 Jada、Jess、Jazmin、Jorge 和 Jamal。队头的字符串是 Jada，`getFront` 可以获取它。方法 `dequeue` 再次获取 Jada 并从队列中删除它（图 7-2f）。随后调用 `enqueue`，将 Jerry 添加到队尾但不会影响队头（图 7-2g）。所以 `getFront` 得到 Jess，且 `dequeue` 获取 Jess 并删除它（图 7-2h）。

现在，如果我们重复执行 `dequeue` 直到队列为空，则再调用 `dequeue` 或是 `getFront` 时都会抛出 `EmptyQueueException`。



**学习问题 1** 执行下列 9 条语句后，队头的字符串是什么？队尾的字符串是什么？

```

QueueInterface<String> myQueue = new LinkedQueue<>();
myQueue.enqueue("Jada");

```

```

myQueue.enqueue("Jess");
myQueue.enqueue("Jazmin");
myQueue.enqueue("Jorge");
String name = myQueue.dequeue();
myQueue.enqueue(name);
myQueue.enqueue(myQueue.getFront());
name = myQueue.dequeue();

```

学习问题 2 定义运行时异常的类 EmptyQueueException。

a) enqueue 操作添加了 Jada

Jada

b) enqueue 操作添加了 Jess

Jada

Jess

c) enqueue 操作添加了 Jazmin

Jada

Jess

Jazmin

d) enqueue 操作添加了 Jorge

Jada

Jess

Jazmin

Jorge

e) enqueue 操作添加了 Jamal

Jada

Jess

Jazmin

Jorge

Jamal

f) dequeue 操作获取并删除了 Jada

Jada

Jess

Jazmin

Jorge

Jamal

g) enqueue 操作添加了 Jerry

Jess

Jazmin

Jorge

Jamal

Jerry

h) dequeue 操作获取并删除了 Jess

Jess

Jazmin

Jorge

Jamal

Jerry

图 7-2 各操作对字符串队列的影响

**!** 程序设计技巧：像 getFront 和 dequeue 这样的方法，当队列为空时的动作必须合理。这里我们规定它们抛出异常。与第 5 章讨论 ADT 栈时一样，也可以定义为返回 null，或给这些方法增加一个队列不能为空的前置条件。但是不推荐给公有方法设定前置条件，这一条在第 5 章段 5.2 的设计决策中提到过。

### 问题求解：模拟排队



日常的许多情况中你都会排队。不管这个队是在商店、售票窗口，还是洗车，排队行为都很像是 ADT 队列。排在队头的人先得到服务；新来者站到队尾，如图 7-3 所示。本问题中，我们将用计算机模拟排队情况。

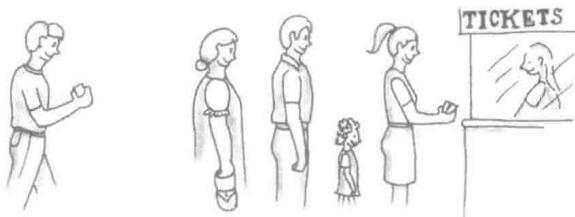


图 7-3 人们排队，或队列

7.5

大多数商家都很关注他们的顾客等待服务的时间。短暂的等待时间，使企业能够提高顾客的满意度，服务更多的人，且赚更多的钱。如果两位代理服务于同一个队列，比起只有一位代理的服务，你等待的时间会更少。但是，商家不想雇佣更多不必要的人员。洗车店肯定不会建立另外一个服务台，来测试顾客排成一队时所等待的时间。

用计算机模拟真实情况，是一种测试不同商业模式的常用方法。本例中，我们将模拟排队的人们等待一位代理服务的情况。顾客以不同的间隔到达，且完成交易所需的时间也不同。假定事件是随机的就能获得这种差异性。

在时间驱动模拟 (time-driven simulation) 中，用一个计数器数出模拟的时间单位——例如分钟。模拟过程中，顾客在随机时间到达并进入队列。为每位顾客指定一个随机的交易时间——即顾客交易所需的时间量——它不会超出某个任意上限。模拟过程中，记下每位顾客在队列中等待的时间。模拟结束时生成汇总统计数据，包括服务的顾客数和顾客平均的等候时间。

7.6

**解决方案设计。**本问题的描述中出现了两类对象：排队和顾客。我们可以为每一种对象设计一个类。

类 `WaitLine` 在给定的一段时间内模拟排队。这段时间内，顾客以随机间隔入队，在得到服务后离队。模拟结束时，由类来计算汇总统计结果。图 7-4 显示了这个类的 CRC 卡。

类 `Customer` 记录并提供顾客的到达时间、交易时间和顾客数。图 7-5 是 `WaitLine` 和 `Customer` 的类图。

| WaitLine |                               |
|----------|-------------------------------|
| 职责       | 模拟顾客排队和离开                     |
|          | 显示服务人数、总的等待时间、平均等待时间和队列中剩下的人数 |
| 协作       |                               |
|          | 顾客                            |
|          |                               |
|          |                               |

图 7-4 类 `WaitLine` 的 CRC 卡



图 7-5 `WaitLine` 和 `Customer` 的类图

7.7

**方法 `simulate`。**方法 `simulate` 是这个例子的核心部分，也是类 `WaitLine` 的主要方法。为维护这个时间驱动模拟时钟，`simulate` 中有一个循环，它对一个给定的时长进行计数。例如，时钟通过从 0 计数到 60 来模拟一个小时。

在时钟的每个值，方法都去查看当前顾客是否仍接受服务，且是否有新来的顾客。如果

有新顾客到来，则方法创建一个新的顾客对象，给对象分配一个随机的交易时间，将顾客放到队列中。如果仍有一位顾客正在接受服务，则时钟前进；如果没有，则一位顾客离开队头并开始接受服务。此时，记下顾客等待的时间。图 7-6 展示了部分模拟的队列示例。

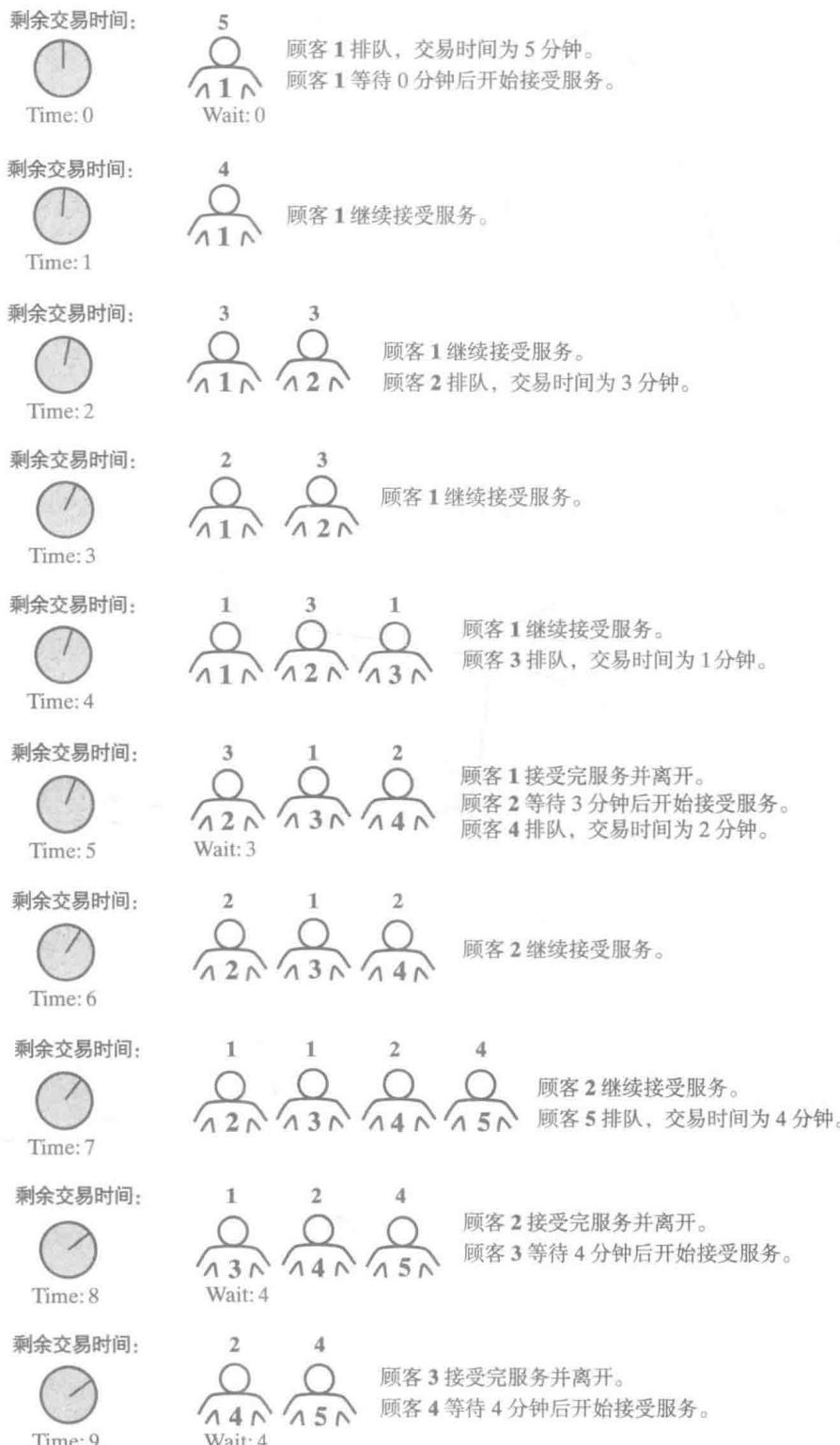


图 7-6 模拟排队

下列伪代码描述方法 `simulate`。它假定类 `WaitLine` 中对数据域进行如下初始化：`line` 是空队列，`numberOfArrivals`、`numberServed` 和 `totalTimeWaited` 均为 0。

```

Algorithm simulate(duration, arrivalProbability, maxTransactionTime)
transactionTimeLeft = 0
for (clock = 0; clock < duration; clock++)
{
 if (新来一名顾客)
 {
 numberOfArrivals++
 transactionTime = 不大于 maxTransactionTime 的随机时间
 nextArrival = 有 clock 和 transactionTime 的新顾客, 顾客号是
 numberOfArrivals
 line.enqueue(nextArrival)
 }
 if (transactionTimeLeft > 0) // If present customer is still being served
 transactionTimeLeft--
 else if (!line.isEmpty())
 {
 nextCustomer = line.dequeue()
 transactionTimeLeft = nextCustomer.getTransactionTime() - 1
 timeWaited = clock - nextCustomer.getArrivalTime()
 totalTimeWaited = totalTimeWaited + timeWaited
 numberServed++
 }
}

```



### 学习问题 3 考虑图 7-6 所示的模拟。

- 什么时候顾客 4 完成服务并离开？
- 顾客 5 开始交易之前等了多长时间？

7.8

`simulate` 的实现细节。在时钟的每个值，`simulate` 都必须判定是否有新顾客到来。为此，它需要顾客的到达概率。这个到达概率是方法的参数，其值为 0 ~ 1。例如，如果顾客在某一给定时刻到来的概率有 65%，则到达概率是 0.65。我们可以使用 Java 语言 `Math` 类中的方法 `random` 生成 0 ~ 1 之间的随机数。如果 `Math.random()` 返回的值小于给定的到达概率，则 `simulate` 创建一个新顾客并放到队列中。

方法给每个新顾客指定一个随机交易时间。给定交易时间的最大值，可以将它乘以 `Math.random()`，得到一个随机时间。结果加 1，以确保交易时间永远不会为 0，但允许极少的情况下交易时间比给定的最大值大 1。为简单起见，我们容忍这个小小的不精确。

类 `WaitList` 的实现列在程序清单 7-2 中。方法 `simulate` 的定义含有打印语句，以帮助你跟踪模拟过程。类中其他的方法都很简单。

#### 程序清单 7-2 类 `WaitLine`

```

1 /** Simulates a waiting line. */
2 public class WaitLine
3 {
4 private QueueInterface<Customer> line;
5 private int numberOfArrivals;
6 private int numberServed;
7 private int totalTimeWaited;
8
9 public WaitLine()
10 {
11 line = new LinkedQueue<>();
12 }
13
14 public void simulate(int duration, double arrivalProbability, int maxTransactionTime)
15 {
16 int clock = 0;
17 int transactionTimeLeft = maxTransactionTime;
18 int timeWaited = 0;
19 int totalWaited = 0;
20 int numberServed = 0;
21 int numberArrivals = 0;
22
23 while (clock < duration)
24 {
25 if (Math.random() < arrivalProbability)
26 {
27 Customer nextCustomer = new Customer();
28 nextCustomer.setArrivalTime(clock);
29 nextCustomer.setTransactionTime(maxTransactionTime * Math.random() + 1);
30 line.enqueue(nextCustomer);
31 numberArrivals++;
32 }
33
34 if (transactionTimeLeft > 0)
35 {
36 transactionTimeLeft--;
37 }
38 else if (!line.isEmpty())
39 {
40 Customer currentCustomer = line.dequeue();
41 transactionTimeLeft = currentCustomer.getTransactionTime() - 1;
42 timeWaited = clock - currentCustomer.getArrivalTime();
43 totalWaited += timeWaited;
44 numberServed++;
45 }
46 }
47 }
48
49 public void printReport()
50 {
51 System.out.println("Number of arrivals: " + numberArrivals);
52 System.out.println("Number of served: " + numberServed);
53 System.out.println("Total time waited: " + totalWaited);
54 }
55
56 public static void main(String[] args)
57 {
58 WaitLine waitLine = new WaitLine();
59 waitLine.simulate(10, 0.5, 5);
60 waitLine.printReport();
61 }
62 }

```

```

12 reset();
13 } // end default constructor
14
15 /** Simulates a waiting line with one serving agent.
16 @param duration The number of simulated minutes.
17 @param arrivalProbability A real number between 0 and 1, and the
18 probability that a customer arrives at
19 a given time.
20 @param maxTransactionTime The longest transaction time for a
21 customer. */
22 public void simulate(int duration, double arrivalProbability,
23 int maxTransactionTime)
24 {
25 int transactionTimeLeft = 0;
26 for (int clock = 0; clock < duration; clock++)
27 {
28 if (Math.random() < arrivalProbability)
29 {
30 numberOfArrivals++;
31 int transactionTime = (int)(Math.random() * maxTransactionTime + 1);
32 Customer nextArrival = new Customer(clock, transactionTime,
33 numberOfArrivals);
34 line.enqueue(nextArrival);
35 System.out.println("Customer " + numberOfArrivals +
36 " enters line at time " + clock +
37 ". Transaction time is " + transactionTime);
38 } // end if
39
40 if (transactionTimeLeft > 0)
41 transactionTimeLeft--;
42 else if (!line.isEmpty())
43 {
44 Customer nextCustomer = line.dequeue();
45 transactionTimeLeft = nextCustomer.getTransactionTime() - 1;
46 int timeWaited = clock - nextCustomer.getArrivalTime();
47 totalTimeWaited = totalTimeWaited + timeWaited;
48 numberServed++;
49 System.out.println("Customer " + nextCustomer.getCustomerNumber() +
50 " begins service at time " + clock +
51 ". Time waited is " + timeWaited);
52 } // end if
53 } // end for
54 } // end simulate
55
56 /** Displays summary results of the simulation. */
57 public void displayResults()
58 {
59 System.out.println();
60 System.out.println("Number served = " + numberServed);
61 System.out.println("Total time waited = " + totalTimeWaited);
62 double averageTimeWaited = ((double)totalTimeWaited) / numberServed;
63 System.out.println("Average time waited = " + averageTimeWaited);
64 int leftInLine = numberOfArrivals - numberServed;
65 System.out.println("Number left in line = " + leftInLine);
66 } // end displayResults
67
68 /** Initializes the simulation. */
69 public final void reset()
70 {
71 line.clear();
72 numberOfArrivals = 0;
73 numberServed = 0;
74 totalTimeWaited = 0;
75 } // end reset
76 } // end WaitLine

```

**7.9****示例输出。Java语句**

```
WaitLine customerLine = new WaitLine();
customerLine.simulate(20, 0.5, 5);
customerLine.displayResults();
```

模拟了 20 分钟的排队过程，其到达概率为 50%，最长交易时间为 5 分钟。得到的结果如下。

```
Customer 1 enters line at time 0. Transaction time is 4
Customer 1 begins service at time 0. Time waited is 0
Customer 2 enters line at time 2. Transaction time is 2
Customer 3 enters line at time 4. Transaction time is 1
Customer 2 begins service at time 4. Time waited is 2
Customer 4 enters line at time 6. Transaction time is 4
Customer 3 begins service at time 6. Time waited is 2
Customer 4 begins service at time 7. Time waited is 1
Customer 5 enters line at time 9. Transaction time is 1
Customer 6 enters line at time 10. Transaction time is 3
Customer 5 begins service at time 11. Time waited is 2
Customer 7 enters line at time 12. Transaction time is 4
Customer 6 begins service at time 12. Time waited is 2
Customer 8 enters line at time 15. Transaction time is 3
Customer 7 begins service at time 15. Time waited is 3
Customer 9 enters line at time 16. Transaction time is 3
Customer 10 enters line at time 19. Transaction time is 5
Customer 8 begins service at time 19. Time waited is 4
Number served = 8
Total time waited = 16
Average time waited = 2.0
Number left in line = 2
```

因为这个示例使用了随机数，所以再次执行 Java 语句时，可能会得到不同的输出结果。

**注：伪随机数**

Java 的方法 `Math.random` 生成均匀分布在 0 ~ 1 之间的数。但是，处理顾客交易的实际时间并不是均匀分布的。它们非常接近，很少有几个会远离平均交易时间。这样的分布称为泊松分布（Poisson distribution）。理想情况下，这个模拟应该使用一个不同的伪随机数生成器。但因我们的最长交易时间值很小，所以使用 `Math.random` 对平均等待时间的影响可能不大。

**问题求解：计算股票售出的资本收益**

假定你买了每股  $d$  美元的  $n$  股股票或是共同基金。之后卖掉了一些股票。如果卖价高于购买价，则你会有收益——资本收益（capital gain）。另一方面，如果卖价低于购买价，则你有损失。我们指定损失为负资本收益。

一般地，投资者在一段时期内会购买特定公司的股票或基金。例如，假定去年你以每股 45 美元的价格购买了 20 股 Presto Pizza。上个月，你以每股 75 美元的价格又购买了 20 股，今天你以每股 65 美元的价格卖掉了 30 股。你的资本收益是多少？并且，实际上你卖的是 40 股中的哪些股？不幸的是，你不能挑选。当计算资本收益时，必须假定，你要按照购买股票的次序来卖它们（就是说，股票销售是先进先出的应用）。故在我们的例子中，你卖掉 20 股以每股 45 美元购买的股票，及 10 股以每股 75 美元购买的股票。30 股的费用是 1650 美元。卖掉它们的费用是 1950 美元，获利 300 美元。

设计一个方法，按时间记录你的投资交易，并计算股票销售的资本收益。

方案设计。为简化示例，我们假定所有的交易都是同一家公司的股票，且它们没有交易手续费。类 StockPurchase 记录一股股票的费用。

图 7-7 是类 StockLedger 的 CRC 卡。使用这个类可以记录按时间购买股票的情况。卖出时，这个类计算资本收益并更新股票持有人的记录。这后两步是相关的，所以，我们将它们放到一个方法中。所以类有两个方法 buy 和 sell，如图 7-8 所示。

下列语句展示了如何使用 StockLedger 来记录问题描述中给定的交易的：

```
StockLedger myStocks = new StockLedger();
myStocks.buy(20, 45); // Buy 20 shares at $45
myStocks.buy(20, 75); // Buy 20 shares at $75
double capGain = myStocks.sell(30, 65); // Sell 30 shares at $65
```

| StockLedger      |
|------------------|
| 职责               |
| 按时间顺序记录购买的股票     |
| 删除卖出的股票，从最长持有的开始 |
| 计算股票售出的资本收益      |
| 协作               |
| 股票               |

图 7-7 类 StockLedger 的 CRC 卡



图 7-8 类 StockLedger 和 StockPurchase 的类图

实现。本例中，StockLedger 将 StockPurchase——它表示我们拥有的股票——的实例记在队列中。队列按时间顺序记录股票，所以我们可以按照购买的次序卖出它们。方法 buy 就是将已购买的股票入队列。

方法 sell 从队列中删除卖出的股票数。操作的同时，它根据卖出价计算总的资本收益并返回这个值。类 StockLedger 列在程序清单 7-3 中。

### 程序清单 7-3 类 StockLedger

```

1 /** A class that records the purchase and sale of stocks, and provides the
2 * capital gain or loss. */
3 public class StockLedger
4 {
5 private QueueInterface<StockPurchase> ledger;
6
7 public StockLedger()
8 {
9 ledger = new LinkedQueue<>();
10 } // end default constructor
11
12 /** Records a stock purchase in this ledger.
13 @param sharesBought The number of shares purchased.

```

```

14 @param pricePerShare The price per share. */
15 public void buy(int sharesBought, double pricePerShare)
16 {
17 while (sharesBought > 0)
18 {
19 StockPurchase purchase = new StockPurchase(pricePerShare);
20 ledger.enqueue(purchase);
21 sharesBought--;
22 } // end while
23 } // end buy
24
25 /** Removes from this ledger any shares that were sold
26 and computes the capital gain or loss.
27 @param sharesSold The number of shares sold.
28 @param pricePerShare The price per share.
29 @return The capital gain (loss). */
30 public double sell(int sharesSold, double pricePerShare)
31 {
32 double saleAmount = sharesSold * pricePerShare;
33 double totalCost = 0;
34
35 while (sharesSold > 0)
36 {
37 StockPurchase share = ledger.dequeue();
38 double shareCost = share.getCostPerShare();
39 totalCost = totalCost + shareCost;
40 sharesSold--;
41 } // end while
42
43 return saleAmount - totalCost; // Gain or loss
44 } // end sell
45 } // end StockLedger

```

## 7.12

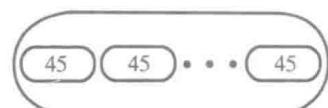
评论下这个方案。典型的股票交易涉及多股股票，两个方法 `buy` 和 `sell` 在它们的参数中反映了这个实际情况。例如，调用 `myStocks.buy(30, 45)` 表示以每股 45 美元购买了 30 股。不过，注意到 `buy` 的实现中，将 30 股一股股地加入队列中。图 7-9a 显示了这样一个队列。这个方法的好处是，`sell` 可以按需删除任意数量的股票。

假定我们将购买的 30 股股票封装为一个对象，并将它加入队列中，如图 7-9b 所示。然后，如果卖掉其中的 20 股，则应该从队列中删除这个对象，并获知股票的购买价格。但我们应该还有 10 股必须保留在队列中。因为它们是最早购买的股票，所以不能简单地将它们添加到队尾；它们必须保留在队头。ADT 队列没有能修改队头项的操作，也没有能在队头添加一项的操作。但如果每个项都有设置方法，则 Java 能允许客户使用 `getFront` 方法返回的引用来修改队头项。这种情形下，你不用删除队头项，直到你卖掉它表示的全部股票为止。本章末尾的练习 10 要求你研究这个方法。

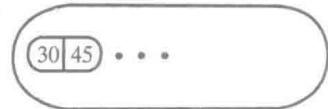
另一方面，如果每个项没有设置方法，则你不能修改它。另外如果每个项表示一股以上的股票，则队列就不是要使用的合适的 ADT。段 7.14 研究了可用的另一种 ADT。



**注：**有设置方法的类是可变对象（mutable object）的类。没有设置方法的类是不可变对象（immutable object）的类。Java 插曲 6 详细讨论了这样的类。



a) 股票单股放在队列中



b) 股票成组作为对象放在队列中

图 7-9 股票在队列中的两种表示

## Java 类库：接口 Queue

Java 类库中的标准包 `java.util` 含有接口 `Queue`，它类似于我们的 `QueueInterface`，但规范说明了更多的方法。我们选择几个方法头列在这里，它们类似于你在本章所见过的。我们还将它们与我们的方法的不同之处标记出来。全部做了标记的那些方法头，表示在我们的接口中没有规范说明类似的方法。再次说明，`T` 是泛型。添加、删除或是获取项的方法均成对出现。

`public boolean add(T newEntry)`

将新项添加到这个队列的队尾，如果成功则返回真，如果不成功则抛出一个异常。

`public boolean offer(T newEntry)`

将新项添加到这个队列的队尾，根据操作成功与否返回真或假。

`public T remove()`

删除并返回这个队列的队头项，如果操作之前队列是空的，则抛出 `NoSuchElementException`。

`public T poll()`

删除并返回这个队列的队头项，如果操作之前队列是空的，则返回 `null`。

`public T element()`

返回这个队列的队头项，如果队列为空，则抛出 `NoSuchElementException`。我们的方法 `getFront` 抛出 `EmptyQueueException` 而不是 `NoSuchElementException`。

`public T peek()`

返回这个队列的队头项，如果队列为空，则返回 `null`。

`public boolean isEmpty()`

检测这个队列是否为空。

`public void clear()`

从这个队列中删除所有的项。

`public int size()`

得到这个队列中当前的元素个数。

这些方法有些是成对出现的。`add` 和 `offer` 都将新项添加到队列中。如果操作不成功，则 `add` 抛出异常，但 `offer` 返回假。同样，方法 `remove` 和 `poll` 都删除并返回队列中的队头项。如果在操作之前队列为空，则 `remove` 抛出异常而 `poll` 返回 `null`。最后，`peek` 和 `element` 都返回队列中的队头项。如果队列为空，则 `element` 抛出异常，而 `peek` 返回 `null`。

可以参考在线文档，详细了解有关 `Queue` 接口。

## ADT 双端队列

假定你正在邮局排队。当轮到你时，邮政代理要求你填一张表。你靠边去填表，而让代

7.14

理为下一位排队的顾客服务。填完表后，代理要服务的下一个对象就是你。实际上，你排到了队头而不是排两次队。

类似地，假如你去排队，然后发现队伍很长，所以决定离开。为了模拟这些例子，需要一个 ADT，它的操作能让你在队列的头尾两端进行添加、删除或取值。这样的 ADT 称为双端队列（double-ended queue，deque，念做“deck”）。

双端队列有与队列一样的操作和与栈一样的操作。例如，双端队列的操作 `addToFront` 和 `removeFront` 分别类似于队列的操作 `enqueue` 和 `dequeue`。而 `addToBack` 和 `removeBack` 分别与栈的 `push` 和 `pop` 操作一样。另外，双端队列有操作 `getFront`、`getBack` 和 `addToFront`。图 7-10 说明了一个双端队列及这些操作。

 注：虽然 ADT deque 称为双端队列，但实际上它的行为更像是双端栈。如图 7-10 所示，你可以在它的两端入栈、出栈或取值。



图 7-10 双端队列示例

因为双端队列操作的规范说明类似于你见过的队列和栈的说明，所以我们简化了程序清单 7-4 的 Java 接口中的注释。

#### 程序清单 7-4 ADT 双端队列的接口

```

1 /**
2 * An interface for the ADT deque.
3 */
4 public interface DequeInterface<T>
5 {
6 /** Adds a new entry to the front/back of this deque.
7 * @param newEntry An object to be added. */
8 public void addToFront(T newEntry);
9 public void addToBack(T newEntry);
10
11 /** Removes and returns the front/back entry of this deque.
12 * @return The object at the front/back of the deque.
13 * @throws EmptyQueueException if the deque is empty before the
14 * operation. */
15 public T removeFront();
16 public T removeBack();
17
18 /** Retrieves the front/back entry of this deque.
19 * @return The object at the front/back of the deque.
20 * @throws EmptyQueueException if the deque is empty. */
21 public T getFront();
22 public T getBack();
23
24 /** Detects whether this deque is empty.
25 * @return True if the deque is empty, or false otherwise. */
26 public boolean isEmpty();
27
28 /* Removes all entries from this deque. */
29 public void clear();
30 } // end DequeInterface

```

栈、队列及双端队列提供的对项的添加、删除及取值操作的对比列在图 7-11 中。

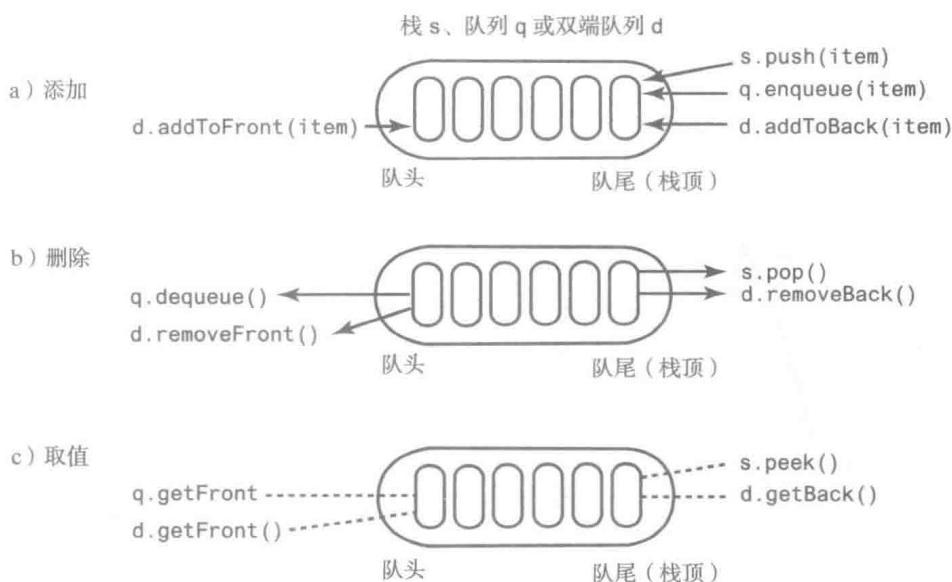


图 7-11 栈 s、队列 q 和双端队列 d 的操作比较



**学习问题 4** 执行下列 9 条语句后，双端队列队头的字符串是什么？队尾的字符串是什么？

```
DequeInterface<String> myDeque = new LinkedDeque<>();
myDeque.addToFront("Jim");
myDeque.addToBack("Jess");
myDeque.addToFront("Jill");
myDeque.addToBack("Jane");
String name = myDeque.getFront();
myDeque.addToBack(name);
myDeque.removeFront();
myDeque.addToFront(myDeque.removeBack());
```

 **示例。**当在键盘上打字时，可能会打错。如果回退以修正错误，那么用什么逻辑能达到你想要的？例如，如果符号←表示回退，则键入

cm ← ompte ←← utr ← er

则得到的结果应该是

computer

每个回退键擦去键入的前一个字符。

7.15

为重复这个过程，当输入字符时，将它们保留在 ADT 中。我们想让这个 ADT 能与栈一样，这样我们可以访问最后键入的字符。但是因为最终我们想让修正后的字符按键入的次序出现，所以想让 ADT 的行为也要与队列一样。ADT 双端队列可以满足这些需求。

下列伪代码使用双端队列来读入并显示从键盘输入的一行：

```
// 读入一行
d = 新的空双端队列
while (没到行尾)
{
 character = 读入的下一个字符
```

```

if (character == ←)
 d.removeBack()
else
 d.addToBack(character)
}
// 显示正确的行
while (!d.isEmpty())
 System.out.print(d.removeFront())
System.out.println()

```

## 问题求解：计算股票售出的资本收益



当总结资本收益示例的讨论时，段 7.12 说明，我们的队列中包含的是一股股的股票。因为股票交易一般都涉及多股，所以将一次交易表示为一个对象更加自然。但你见到的，如果我们使用队列，则交易对象必须要有设置方法。如果我们使用双端队列，那就不是这种情况了。

7.16

本节，我们修改段 7.10 介绍的类 StockLedger 的实现，但不修改它的设计。我们还要修改类 StockPurchase，让其表示以每股  $d$  美元购买的  $n$  股股票，如段 7.12 中提到的。修改后的类有数据域 shares 和 cost、一个构造方法及访问方法 getNumberOfShares 和 getCostPerShare。

我们可以修改程序清单 7-3 中给出的 StockLedger 的实现如下。数据域 ledger 现在是双端队列的实例而不是队列的实例。方法 buy 创建 StockPurchase 的实例，并将它放到双端队列的队尾，如下所示。

```

public void buy(int sharesBought, double pricePerShare)
{
 StockPurchase purchase = new StockPurchase(sharesBought, pricePerShare);
 ledger.addToBack(purchase);
} // end buy

```

方法 sell 更加复杂。它必须从双端队列的队头删除 StockPurchase 对象，并判定这个对象表示的股票数是否多于所卖掉的数。如果是，则方法创建一个新的 StockPurchase 实例，来表示剩余的股票。

然后将实例添加到双端队列的队头，因为这些股票应该是下一次被卖掉的。

```

public double sell(int sharesSold, double pricePerShare)
{
 double saleAmount = sharesSold * pricePerShare;
 double totalCost = 0;
 while (sharesSold > 0)
 {
 StockPurchase transaction = ledger.removeFront();
 double shareCost = transaction.getCostPerShare();
 int numberofShares = transaction.getNumberOfShares();
 if (numberofShares > sharesSold)
 {
 totalCost = totalCost + sharesSold * shareCost;
 int numberToPutBack = numberofShares - sharesSold;
 StockPurchase leftOver = new StockPurchase(numberToPutBack,
 shareCost);
 ledger.addToFront(leftOver); // Return leftover shares
 // Note: Loop will exit since sharesSold will be <= 0 later
 }
 else

```

```

 totalCost = totalCost + numberOfShares * shareCost;
 sharesSold = sharesSold - numberOfShares;
} // end while
return saleAmount - totalCost; // Gain or loss
} // end sell
}

```

## Java 类库：接口 Deque

Java 类库的标准包 `java.util` 中含有一个接口 `Deque`, 它类似于我们的 `DequeInterface`, 但规范说明了更多的方法。这里我们选择这个接口所声明的几个方法头。添加、删除或是取值方法都是成对出现的。如果操作不成功, 其中的一个方法抛出一个异常, 而另一个方法返回 `null` 或是假值。`T` 是双端队列中项的泛型。

`public void addFirst(T newEntry)`

将新项添加到该双端队列的队头, 但如果不能, 则抛出几个异常中的一个。

`public boolean offerFirst(T newEntry)`

将新项添加到该双端队列的队头, 根据操作成功与否返回真或假。

`public void addLast(T newEntry)`

将新项添加到该双端队列的队尾, 但如果不能, 则抛出几个异常中的一个。

`public boolean offerLast(T newEntry)`

将新项添加到该双端队列的队尾, 根据操作成功与否返回真或假。

`public T removeFirst()`

删除并返回该双端队列的队头项, 但如果操作前双端队列为空, 则抛出异常 `NoSuchElementException`。

`public T pollFirst()`

删除并返回该双端队列的队头项, 但如果操作前双端队列为空, 则返回 `null`。

`public T removeLast()`

删除并返回该双端队列的队尾项, 但如果操作前双端队列为空, 则抛出异常 `NoSuchElementException`。

`public T pollLast()`

删除并返回该双端队列的队尾项, 但如果操作前双端队列为空, 则返回 `null`。

`public T getFirst()`

返回该双端队列的队头项, 但如果双端队列为空, 则抛出异常 `NoSuchElementException`。

`public T peekFirst()`

返回该双端队列的队头项, 但如果双端队列为空, 则返回 `null`。

`public T getLast()`

返回该双端队列的队尾项，但如果双端队列为空，则抛出异常 `NoSuchElementException`。

`public T peekLast()`

返回该双端队列的队尾项，但如果双端队列为空，则返回 `null`。

`public boolean isEmpty()`

检测该双端队列是否为空。

`public void clear()`

删除该双端队列中的所有项。

`public int size()`

得到该双端队列中当前的项数。

接口 `Deque` 派生于接口 `Queue`，所以它还有之前在段 7.13 中描述过的方法 `add`、`offer`、`remove`、`poll`、`element` 和 `peek`。另外，`Deque` 声明了如下的两个栈方法：

```
public void push(T newEntry)
public T pop()
```

除了 `push` 在 `Deque` 中是一个 `void` 方法之外，上述这些方法与之前我们在第 5 章段 5.23 中见过的 `java.util.Stack` 类中定义的方法是一样的。正如我们在第 5 章提到过的，你不应该再使用标准类 `Stack`。下一段描述了你应该使用的类。

Java 类库中接口 `Deque` 的在线文档，列出了对应于队列方法及栈方法的双端队列中的方法。

## Java 类库：类 `ArrayDeque`

**7.18** Java 类库的标准包 `java.util` 中含有类 `ArrayDeque`，它实现了我们刚描述的接口 `Deque`。因为 `Deque` 声明了对应于双端队列、队列和栈的方法，所以你可以使用 `ArrayDeque` 来创建这其中任一种数据集合的示例。

下面两个构造方法是这个类定义的：

`public ArrayDeque()`

创建初始时有 16 个项的空双端队列。

`public ArrayDeque(int initialCapacity)`

创建一个带给定的初始容量的空双端队列。

`ArrayDeque` 实例的大小可由客户按需增长。



**注：**如第 5 章结尾处的“注”所表示的，如果你想使用标准类而不是你自己的类去创建一个栈，那么你应该使用标准类 `ArrayDeque` 而不是标准类 `Stack` 的实例。

`ArrayDeque` 是新的类，它提供的栈的实现比 `Stack` 更快。`Stack` 仍保留在 Java 类库中，以支持之前已写的 Java 程序。

## ADT 优先队列

虽然银行按顾客到来的次序提供服务，但急诊室根据病人病症的紧急程度为病人诊疗。银行按时间顺序使用队列来组织顾客。医院为每位病人指定一个优先级 (priority)，它比病人到达的时间更重要。

ADT 优先队列 (priority queue) 根据对象的优先级组织它们。具体是哪种优先级要依赖于对象的属性。例如，优先级可以是整数。优先级 1 可能是最高优先级，也可能是最低优先级。让对象是 Comparable 的，我们可以将这个细节隐藏在对象的 compareTo 方法中。则优先队列可以使用 compareTo 来比较对象的优先级。所以优先队列可以有程序清单 7-5 中所给的 Java 接口。我们使用记号 ? super T 来表示泛型 T 的任何父类。段 J5.13 中将这个记号用在另一个例子中。

**程序清单 7-5** ADT 优先队列的接口

```

1 public interface PriorityQueueInterface<T extends Comparable<? super T>>
2 {
3 /** Adds a new entry to this priority queue.
4 @param newEntry An object to be added. */
5 public void add(T newEntry);
6
7 /** Removes and returns the entry having the highest priority.
8 @return Either the object having the highest priority or, if the
9 priority queue is empty before the operation, null. */
10 public T remove();
11
12 /** Retrieves the entry having the highest priority.
13 @return Either the object having the highest priority or, if the
14 priority queue is empty, null. */
15 public T peek();
16
17 /** Detects whether this priority queue is empty.
18 @return True if the priority queue is empty, or false otherwise. */
19 public boolean isEmpty();
20
21 /** Gets the size of this priority queue.
22 @return The number of entries currently in the priority queue. */
23 public int getSize();
24
25 /** Removes all entries from this priority queue. */
26 public void clear();
27 } // end PriorityQueueInterface

```



### 设计决策：哪个 ADT 可以有 null 数据？

在第 5 章段 5.2 中的设计决策中，我们决定，返回值 null 或是表示方法失败，或是表示集合中的一个有效项，但不能兼顾。它还提到，本书中大多数 ADT 允许 null 值作为有效的数据项。栈、队列和双端队列都是这样的 ADT。如果试图从空的数据集中获取或删除一个项时，它们的方法会抛出一个异常。根据与项值无关的评判准则而决定项是无序或有序的任何一个 ADT，都可以含有 null 的项。例如，包中的数据是无序的，而栈或队列中的数据是按其添加到 ADT 中的顺序而定的，双端队列中的数据是按它何时及添加在哪端的次序而定的。这些 ADT 中的每一个都能有 null 项。但是优先队列，基于项的优先级的数值通过比较来决定它们的次序。它不能有 null 的项。所以从优先队列中删除或获取一项时，若失败可用 null 表示。



**学习问题 5** 执行下列语句后，优先队列的队头是哪个字符串？队尾是哪个字符串？字符串的字典序决定它们的优先级。注意，“z”比“a”的优先级更高。

```
PriorityQueueInterface<String> myPriorityQueue = new LinkedPriorityQueue<>();
myPriorityQueue.add("Jane");
myPriorityQueue.add("Jim");
myPriorityQueue.add("Jill");
String name = myPriorityQueue.remove();
myPriorityQueue.add(name);
myPriorityQueue.add("Jess");
```

## 问题求解：跟踪指派



教授和老板喜欢在特定的日期给我们分配任务。使用优先队列，将这些分配按应该完成的顺序组织起来。

7.20

为使示例简单，我们按到期日期对任务进行排序。有最早到期日期的任务有最高的优先级。我们可以对任务定义一个类 `Assignment`，它包含日期域 `date`，表示任务的到期日期。图 7-12 是这种类的类图。我们假定 `date` 是 `Comparable` 类的一个实例，比如 Java 类库中的 `java.sql.Date`。所以，如果 `date` 早于 `otherDate`，则表达式 `date.compareTo(otherDate)` 是负的。`Assignment` 中的 `compareTo` 方法如下：

```
public int compareTo(Assignment other)
{
 return date.compareTo(other.date);
} // end compareTo
```

| Assignment      |
|-----------------|
| course—课程代码     |
| task—指派描述       |
| date—截止日期       |
| getCourseCode() |
| getTask()       |
| getDueDate()    |
| compareTo()     |

图 7-12 类 `Assignment` 的类图

更复杂版本的 `Assignment` 可以在 `compareTo` 中包含评估优先级的其他标准。



### 注：类 `java.sql.Date`

Java 类库的包 `java.sql` 中的类 `Date` 有一个构造方法，它的参数将日期指定为从 GMT 时间 1970 年 1 月 1 日午夜开始算起的微秒数。构造 `Date` 对象更方便的方法是使用下列静态方法 `valueOf`：

```
public static Date valueOf(String s)
```

返回一个 `Date` 对象，其值由格式为 `yyyy-mm-dd` 的字符串 `s` 给定。

例如，表达式 `Date.valueOf("2020-02-29")` 返回一个表示 2020 年 2 月 29 日的 `Date` 对象。

`Date` 实现了接口 `Comparable<Date>`，并重写了 `toString`。

7.21

我们可以将 `Assignment` 的实例直接添加到优先队列中，也可以写一个简单的包装类 `AssignmentLog` 来组织我们的任务。如图 7-13 所示，`AssignmentLog` 有一个数据域 `log`，它是优先队列的实例，含有按优先级次序排定的任务。方法 `addProject`、`getNextProject` 和 `removeNextProject` 间接维护优先级队列。

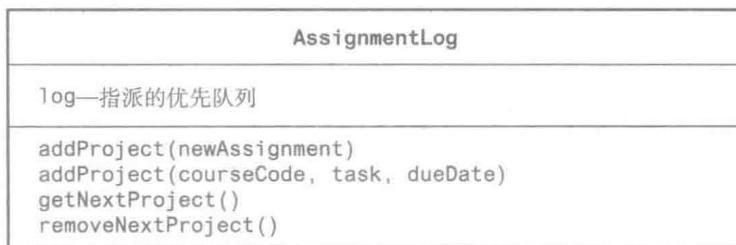


图 7-13 类 AssignmentLog 的类图

AssignmentLog 的实现列在程序清单 7-6 中。

### 程序清单 7-6 类 AssignmentLog

```

1 import java.sql.Date;
2 public class AssignmentLog
3 {
4 private PriorityQueueInterface<Assignment> log;
5
6 public AssignmentLog()
7 {
8 log = new PriorityQueue<>();
9 } // end constructor
10
11 public void addProject(Assignment newAssignment)
12 {
13 log.add(newAssignment);
14 } // end addProject
15
16 public void addProject(String courseCode, String task, Date dueDate)
17 {
18 Assignment newAssignment = new Assignment(courseCode, task, dueDate);
19 addProject(newAssignment);
20 } // end addProject
21
22 public Assignment getNextProject()
23 {
24 return log.peek();
25 } // end getNextProject
26
27 public Assignment removeNextProject()
28 {
29 return log.remove();
30 } // end removeNextProject
31 } // end AssignmentLog

```

AssignmentLog 的客户程序中可能有下列语句:

```

AssignmentLog myHomework = new AssignmentLog();
myHomework.addProject("CSC211", "Pg 50, Ex 2", Date.valueOf("2019-2-20"));
Assignment pg75Ex8 = new Assignment("CSC215", "Pg 75, Ex 8",
 Date.valueOf("2019-3-14"));
myHomework.addProject(pg75Ex8);
.
.
System.out.println("The following assignment is due next:");
System.out.println(myHomework.getNextProject());

```

显示的是有最早到期日期的任务，但不从任务日志中删除。

### Java 类库：类 PriorityQueue

Java 类库的标准包 `java.util` 中含有类 `PriorityQueue`。这个类实现了我们在本章

前面介绍过的接口 Queue。PriorityQueue 的实例与优先队列的行为一样，但与队列不同，其中的项是有序的，具有最小值的项，因此也是最高优先级的项，位于优先队列的最前面。因为 PriorityQueue 使用了方法 compareTo 来排列项，所以项必须属于实现了接口 Comparable 的类。另外，项的值不能是 null。回忆 T 是泛型。

下面是类 PriorityQueue 的基本构造方法和方法。添加、删除或获取方法都是成对出现的。

```
public PriorityQueue()
```

创建一个初始容量有 11 个项的空优先队列。

```
public PriorityQueue(int initialCapacity)
```

创建有给定初始容量的空优先队列。

```
public boolean add(T newEntry)
```

将新项添加到这个优先队列中，如果成功则返回真，否则抛出一个异常。

```
public boolean offer(T newEntry)
```

将新项添加到这个优先队列中，根据操作的成功与否，返回真或假。

```
public T remove()
```

删除并返回优先队列最前面的项，如果操作前优先队列为空，则抛出 NoSuchElementException。

```
public T poll()
```

删除并返回优先队列最前面的项，如果操作前优先队列为空，则返回 null。

```
public T element()
```

返回优先队列最前面的项，如果优先队列为空，则抛出 NoSuchElementException。

```
public T peek()
```

返回优先队列最前面的项，如果优先队列为空，则返回 null。

```
public boolean isEmpty()
```

检测优先队列是否为空。

```
public void clear()
```

删除本优先队列中的所有项。

```
public int size()
```

得到本优先队列中当前的元素个数。

PriorityQueue 的实例数按照客户的需要增大。

## 本章小结

- ADT 队列按先进先出的规则组织项。这些项中，最先或最早添加的项位于队头，最近添加的项位于队尾。

- 队列的主要操作——enqueue、dequeue 和 getFront——仅处理队列的端点处。方法 enqueue 将项添加到队尾；dequeue 删除并返回队头项，而 getFront 只是返回这个项。
- 可以使用队列来模拟排队。时间驱动模拟对模拟的时间单位进行计数。顾客随机到达，且被赋值一个随机交易时间，并进入队列。
- 当计算股票售出的资本收益时，必须按购买股票的次序来卖出它们。如果在队列中将购买的股票单独记录，则它们的次序即是要卖出的次序。
- 双端队列可以在头尾两端进行添加、删除或获取项的操作。如此，它结合并扩展了队列和栈的操作。双端队列的主要操作是 addToFront、removeFront、getFront、addToBack、removeBack 和 getBack。
- 优先队列按项的优先级组织项，优先级按项的 compareTo 方法确定。除了将项添加到优先队列中外，还可以获取并删除有最高优先级的项。
- 可变对象有设置方法；不可变对象没有相应的方法。

## 程序设计技巧

- 像 getFront 和 dequeue 这样的方法，当队列为空时的动作必须合理。这里我们规定它们抛出异常。与第 5 章讨论 ADT 栈时一样，也可以定义为返回 null。因为这些方法是公有的，不推荐给这些方法增加一个队列不能为空的前置条件。

## 练习

1. 如果将对象 x、y 和 z 添加到初始为空的队列中，3 次 dequeue 操作将它们从队列中删除的次序是什么？
2. 如果将对象 x、y 和 z 添加到初始为空的双端队列中，3 次 removeBack 操作将它们从双端队列中删除的次序是什么？
3. 执行下列语句后，队列的内容是什么？

```
QueueInterface<String> myQueue = new LinkedQueue<>();
myQueue.enqueue("Jane");
myQueue.enqueue("Jess");
myQueue.enqueue("Jill");
myQueue.enqueue(myQueue.dequeue());
myQueue.enqueue(myQueue.getFront());
myQueue.enqueue("Jim");
String name = myQueue.dequeue();
myQueue.enqueue(myQueue.getFront());
```

4. 执行下列语句后，双端队列的内容是什么？

```
DequeInterface<String> myDeque = new LinkedDeque<>();
myDeque.addToFront("Jim");
myDeque.addToFront("Jess");
myDeque.addToBack("Jill");
myDeque.addToBack("Jane");
String name = myDeque.removeFront();
myDeque.addToBack(name);
myDeque.addToBack(myDeque.getFront());
myDeque.addToFront(myDeque.removeBack());
myDeque.addToFront(myDeque.getBack());
```

5. 执行下列语句后，优先队列的内容是什么？假定按字典序最早的字符串有最高的优先级。

```

PriorityQueueInterface<String> myPriorityQueue = new LinkedPriorityQueue<>();
myPriorityQueue.add("Jim");
myPriorityQueue.add("Jess");
myPriorityQueue.add("Jill");
myPriorityQueue.add("Jane");
String name = myPriorityQueue.remove();
myPriorityQueue.add(name);
myPriorityQueue.add(myPriorityQueue.peek());
myPriorityQueue.add("Jim");
myPriorityQueue.remove();

```

6. 考虑可以划分的字符串，其前一半与后一半相同（忽略空格、标点符号和大小写）。例如，字符串 "booboo" 可以划分为 "boo" 和 "boo"。另一个例子是 "hello, hello"。忽略空格和逗号后，该字符串的两半是一样的。但是字符串 "rattan" 的两半不相同，字符串 "abcab" 也不同。描述你如何使用队列来测试一个字符串是否有这个特性。
7. 完成图 7-6 中所示的模拟。让顾客 6 在时刻 10 进入队列，交易时间为 2。
8. 假定 `customerLine` 是段 7.8 中所给的类 `WaitLine` 的实例。调用 `customerLine.simulate(15, 0.5, 5)` 产生下列随机事件：  
 顾客 1 在时刻 6 入队，交易时间为 3。  
 顾客 2 在时刻 8 入队，交易时间为 3。  
 顾客 3 在时刻 10 入队，交易时间为 1。  
 顾客 4 在时刻 11 入队，交易时间为 5。  
 模拟过程中，有多少顾客得到服务？他们的平均等待时间是多少？
9. 重做练习 8，但使用下列随机事件：  
 顾客 1 在时刻 0 入队，交易时间为 4。  
 顾客 2 在时刻 1 入队，交易时间为 4。  
 顾客 3 在时刻 3 入队，交易时间为 1。  
 顾客 4 在时刻 4 入队，交易时间为 4。  
 顾客 5 在时刻 9 入队，交易时间为 3。  
 顾客 6 在时刻 12 入队，交易时间为 2。  
 顾客 7 在时刻 13 入队，交易时间为 1。
10. 当使用队列来计算资本收益时，从段 7.12 中内容知道，如果每个项有设置方法，则每个队列项都能表示一股以上的股票。修改类 `StockPurchase`，以便每个实例都有设置方法，且表示一个公司的多股股票。然后修改类 `StockLedger`，使用队列保存 `StockPurchase` 对象。
11. 第 5 章练习 11 描述了一个回文。你能使用本章描述的除栈以外的一个 ADT，来查看一个字符串是否是回文？如果可行，为每一种可用的 ADT 开发一个算法来实现这个思想。
12. 考虑某种栈，有有限的大小但允许无限次地进行 `push` 操作。当进行 `push` 操作时如果栈满了，则从栈底删除一项从而为新项腾出空间。维护有限历史记录的浏览器可以使用这种类型的栈。使用双端队列来实现这个栈。

## 项目

1. 使用 Java 类库中的 `java.util.ArrayDeque`，定义并测试实现了程序清单 7-1 中给出的接口 `QueueInterface` 的类 `OurQueue`。查阅 Java 类库的在线文档了解 `ArrayDeque` 的方法。
2. 使用 Java 类库中的类 `java.util.ArrayDeque`，定义并测试实现了程序清单 7-4 中给出的接口 `DequeInterface` 的类 `OurDeque`。
3. 使用 Java 类库中的 `java.util.PriorityQueue`，定义并测试实现了程序清单 7-5 中给出的接口 `PriorityQueueInterface` 的类 `OurPriorityQueue`。查阅 Java 类库的在线文档了解

PriorityQueue 的方法。

在下列项目中当你需要使用队列、双端队列或是优先队列时，使用项目 1、2 和 3 要求你完成的 OurQueue、OurDeque 和 OurPriorityQueue 类。

4. 扩展本章描述的资本收益示例，允许投资组合中出现多类股票。使用代表股票符号的字符串来标识不同的股票。使用单独的队列、双端队列或优先队列记录每个公司的股票。将这些 ADT 集合保存在向量中。
5. 模拟有一条跑道的小型空港。等待起飞的飞机加入机场上的队列中。等待降落的飞机加入空中的队列中。任何的给定时刻，只有一架飞机可以使用跑道。空中的所有飞机必须在任何飞机起飞前降落。
6. 重复项目 5，但对于等待起飞的飞机使用优先队列。为燃料不足或是有机械故障的模拟过程实现优先级调度算法。
7. 当集合中的每个对象有一个优先级时，应该如何组织有相同优先级的多个对象？一种方法是具有相同优先级的对象按时间顺序组织。所以可以创建一个队列的优先级队列。设计这样一个 ADT。
8. 写一个模拟火车路线的程序。一条火车路线由若干车站组成，起始点和终到点都是一个终端站。给定火车经过线路上两个相邻站之间所需的时间。每个站都关联一个乘客队列。乘客随机生成，随机分配入站，并随机指定一个终到站。火车定期从终端站出发，按路线到达路线中的各车站。当火车停在一个车站时，在那一站下车的所有乘客先退出。然后排在那个车站队列中的任意乘客登上火车，直到或者队列为空或者火车已满时为止。
9. 写一个模拟操作系统中作业调度的程序。作业随机生成。每个作业随机给定一个 1 ~ 4 之间的优先级——其中 1 是最高优先级——及运行它所需的随机时间。

作业的运行没有开始也没有结束，而是共享处理器。操作系统在称为时间片的固定的时间单位内运行一个作业。在时间片结束时，当前作业的运行被挂起。然后作业被放到优先队列中，在那儿等待下一次共享处理器时间。然后有最高优先级的作业从优先队列中删除并在时间片内运行。

当作业首次生成时，如果处理器空闲则它立即开始运行，否则它将被放到优先队列中。

10. int 类型的最大正整数是 2 147 483 647。另一种整数类型 long，能表示的最大数是 9 223 372 036 854 775 807。假定你想表示更大的整数。例如，密码学中使用多于 100 位的整数。设计并实现非常大的非负整数的类 Huge。最大的整数应含有至少 30 位。使用双端队列来表示一个整数值。

为这个类提供的操作有

- 设置非负整数的值（提供设置方法和构造方法）
- 将非负整数值作为字符串返回
- 读入一个大的非负整数（跳过前导 0，但记住 0 是有效数字）
- 显示大的非负整数（不显示前导 0，但如果整数是 0，则显示一个 0）
- 将两个非负整数相加，其和为第三个整数
- 将两个非负整数相乘，其积为第三个整数

当读入、相加或相乘整数时，应该处理溢出。如果整数超出 MAX\_SIZE 位，则太大了，其中 MAX\_SIZE 是你定义的命名常量。写一个测试程序，验证每个方法。

11. 一种洗牌方法是完美洗牌（perfect shuffle）。首先，将 52 张牌分为各含 26 张牌的两半。接下来，按照下面的方法合并这两半。先从上面的一半开始，然后换下面一半，从一半中拿走最下面的牌，将它放到新牌的最上面。

例如，如果我们的牌含有 6 张 1 2 3 4 5 6，上面的一半是 1 2 3，下面的一半是 4 5 6。则位于上面一半的最下面的 3，成为洗牌的最下面一张。然后将 6，即下面一半的最下面一张，放到洗好的牌的最上面。接下来，将 2 放到上面，然后是 5, 1，最后是 4。洗好的牌则是 4 1 5 2 6 3。注意，原来最上面的牌现在是洗好的牌的第二张，原来的最下面的牌现在是洗好的牌从下面数的第二张。

这个洗牌称为一次内洗 (in-shuffle)，这是指从上面一半开始将牌放到洗好的牌中的过程。如果你从下一半开始，则得到外洗 (out-shuffle)，其中，原来最上面的牌和最下面的牌在洗好的牌中，仍维持在原来的位置。

定义扑克游戏的一个类，使用双端队列来保存牌。类应该定义完成完美内洗和完美外洗的方法。使用你的类

- 对于  $n$  张牌，将它又返回原次序所需的完美外洗的次数。
- 对于  $n$  张牌，将它又返回原次序所需的完美内洗的次数。
- 通过执行一系列的内洗和外洗，你可以将最上面的一张牌，其位置为 0，移到任何想要的位置  $m$ ，过程如下。将  $m$  表示为二进制形式。从最左的 1 开始，向右处理，对遇到的每个 1 执行一次内洗，对遇到的每个 0 执行一次外洗。例如，如果  $m$  是 8，其等价的二进制是 1000。我们将执行一次内洗和 3 次外洗，将原来最上面的牌移到位置 8，即这是从上面数第 9 张牌。定义一个方法执行这个纸牌魔术。

12.(财务 / 模拟) 当人们排队等待服务时，比如在超市、影院或是机场，服务提供者希望尽量减少其雇佣的代理的数量，同时将顾客的等待时间保持在可容忍的水平。

- 本地商店的结账区有 8 个付款登记簿，每个登记簿前都排一个队。因为受地方大小所限，每个队伍只能容纳最多 4 个人。结果，当所有的结账队伍都满了时，新的顾客只能不购买就离开。模拟这个结账区。假定每名顾客的平均服务时间是 2 分钟。如果每  $t$  分钟到来一名新顾客，则  $t$  取什么值时，这个商店在 16 小时的工作日内能服务的顾客数最多？且丢失的顾客数最少？
- 现在雇你来重新设计商店的结账区。你明白，当多个代理服务于一个队时，比每个代理都有自己的队时，顾客的平均等待时间更短。模拟一个结账区，仅有一个队，由 8 个付款登记簿提供服务。当 32 个人排队等待时，新来的顾客不买就离开商店了。假定每名顾客的平均服务时间是 2 分钟。如果每  $t$  分钟到来一名新顾客，则  $t$  取什么值时，这个商店能服务的顾客数最多？且丢失的顾客数最少？

13.(模拟) 典型的操作系统 (OS) 在它调度程序访问处理器 (CPU) 时给程序排定优先级。考虑操作系统指定的优先级范围在 1 ~ 64 之间，其中 1 是最高优先级。它们使用不同的算法来调度程序由 CPU 运行。

当程序开始运行时，它不会执行到完成。而是执行一个固定的时间单位，称为时间片。时间片到了，挂起程序的执行，它等待下一次处理器的时间份额。下一个被调度的程序开始执行。

考虑 50 个程序，每个都有一个随机优先级及随机执行时间，后者表示为微秒的一个整数。将程序提交给 OS 调度运行，每次一个。所有的程序都从时间 0 开始等待执行。在 OS 调度完所有程序后，第一个程序开始执行。

模拟下列每个调度算法。哪个算法能提供程序运行的最小平均等待时间？改变时间片的大小——这会改变你的结果吗？

- 第一个操作系统 (OS-1) 使用 4 个队列——A、B、C 和 D——实现其优先调度算法：
  - 队列 A 含有其优先级最高且范围在 1 ~ 15 之间的程序。
  - 队列 B 含有其优先级在 16 ~ 31 之间的程序。
  - 队列 C 含有其优先级在 32 ~ 47 之间的程序。
  - 队列 D 含有其优先级在 48 ~ 64 之间的程序。

每次删除并运行队列 A 中的一个程序，直到队列为空。按照这个模式继续执行队列 B 中的程序，然后是队列 C，最后是队列 D。计算程序在队列中等待的平均时间和总时间。

- OS-2 使用循环赛调度算法，其中每个程序在给定的时间片内访问 CPU。程序的时间片结束后，另一个程序运行它的时间片。这个过程一直继续，直到程序完成其运行时为止。

当在时间片结束程序运行暂停时，OS 必须将其放回相应的队列中。队列 A 中具有最高优先级的程序将回到队头，不过，为此我们需要一个双端队列——deque——而不是队列。所以，使用双

端队列 A 替换队列 A。程序  $p$  从双端队列 A 的队头移走并执行一段时间；然后另一个程序  $q$  从双端队列 A 的队头移走且运行一段时间，而程序  $p$  返回双端队列 A 的队头。实际上，OS 交替执行两个具有最高优先级的程序，直到它们运行完毕。

OS-2 使用优先队列而不是队列 B。当双端队列 A 变为空时，OS-2 考虑优先队列 B 中的程序。当 B 中程序的时间片到了，将程序的优先级值加 1 以降低其优先级。然后 OS 根据新的优先级将程序放回 B 或 C 中。注意，程序新的优先级是 32，则它可能进入队列 C 的队尾。

当 B 为空时，OS-2 考虑队列 C 中的程序。当 C 中程序的时间片到了，程序返回到队尾。当队列 C 为空后，OS-2 考虑队列 D 中具有最低优先级的程序。一旦这样的程序开始执行，会一直执行到完成。

c. OS-3 使用类似于 OS-2 的循环赛调度算法，但让双端队列 A 中的程序执行 2 个时间片，然后一个时间片给优先队列 B，一个时间片给队列 C，两个时间片给双端队列 A，以此类推。仅当 A、B、C 都为空后，OS 才考虑队列 D。

# 队列、双端队列和优先队列的实现

先修章节：第 2 章、第 3 章、第 6 章、第 7 章

## 目标

学习完本章后，应该能够

- 使用结点链表或是数组实现 ADT 队列
- 在双向结点链表的两端添加或删除结点
- 使用双向结点链表实现 ADT 双端队列
- 使用数组或结点链表实现 ADT 优先队列

本章讨论的 ADT 队列的实现用到了实现 ADT 包和 ADT 栈时用过的技术。我们将使用结点链表或是数组来保存队列的项。虽然我们在第 6 章见到过的栈的实现十分简单，但队列的实现涉及的问题更多一些。

我们还介绍双端队列的链式实现。因为允许访问双端队列的前端和后端，所以普通的链式结点就达不到要求了。比如说，没有指向前一个结点的引用，就不能删除链表中的最后结点。所以，我们使用一种新的在两个方向上链接结点的链表。即链表中的一个结点指向后一个结点和前一个结点。这样的链表用来实现双端队列是足够了。

最后，我们提出 ADT 优先队列的一些实现方案。但是注意到，当在第 24 章和第 27 章遇到 ADT 堆时，可能会有更高效的实现。

## 队列的链式实现

如果使用结点链表来实现队列，则队列的两端必须在链表的相对的两端。如果仅有一个指向链表的头引用，则访问链表的最后结点时需要遍历整个的链表，这样访问的效率不高。增加一个尾引用 (tail reference)——指向链表中最后结点的外部引用——是解决这个问题的一种方法，且是我们这里要采用的方法。

使用头引用和尾引用，哪个结点应该是队头，哪个结点应该是队尾？我们必须能从队头删除项。如果它在链表的开头，则删除它很容易。如果它在链表的尾端，删除它则需要一个指向其前一个结点的引用。为得到这个引用，必须遍历链表。所以，我们抛弃这个选择，让链表的首结点中含有队头项。

将队头放到链表的开头，显然就让队尾放到了链表尾。因为我们仅在队尾添加项，且因为我们有用于链表的尾引用，所以这样的安排还是不错的。

图 8-1 图示了有头引用和尾引用的结点链表。队列中的每个项对应于链表中的一个结点。仅当需要一个新项时才分配结点，当删除项时回收结点。



学习问题 1 从结点链表中删除最后一个结点时，为什么尾引用帮不上忙？

有队头项。`lastNode` 指向链表中的最后一个结点，它含有队尾项。因为当队列为空时，这两个域都为 `null`，故默认的构造方法将它们设置为 `null`。类的框架列在程序清单 8-1 中。

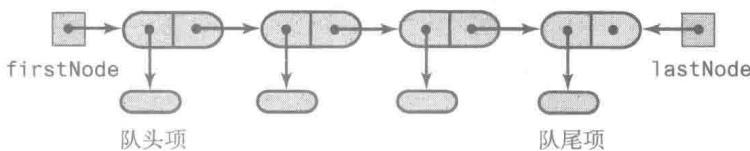


图 8-1 实现队列的结点链表

类中还含有私有类 `Node`，与第 3 章程序清单 3-4 中所见到的一样。我们在第 6 章实现 ADT 栈时也用到过这个类。

### 程序清单 8-1 ADT 队列链式实现的框架

```

1 /**
2 * A class that implements a queue of objects by using
3 * a chain of linked nodes.
4 */
5 public final class LinkedQueue<T> implements QueueInterface<T>
6 {
7 private Node firstNode; // References node at front of queue
8 private Node lastNode; // References node at back of queue
9
10 public LinkedQueue()
11 {
12 firstNode = null;
13 lastNode = null;
14 } // end default constructor
15
16 < Implementation of the queue operations go here. >
17 .
18
19 private class Node
20 {
21 private T data; // Entry in queue
22 private Node next; // Link to next node
23
24 < Constructors and the methods getData, setData, getNextNode, and setNextNode
25 are here. >
26
27 .
28 } // end Node
29 } // end LinkedQueue

```

**在队尾添加。**为了将一个项添加到队尾，需要分配一个新结点，并将它添加到链尾处。8.3 如果队列是空的——所以链表也是空的——则我们要将两个数据域 `firstNode` 和 `lastNode` 都指向新结点，如图 8-2 所示。否则，链表中最后一个结点和数据域 `lastNode` 都必须指向新结点，如图 8-3 所示。



图 8-2 将新结点添加到空链表之前和之后

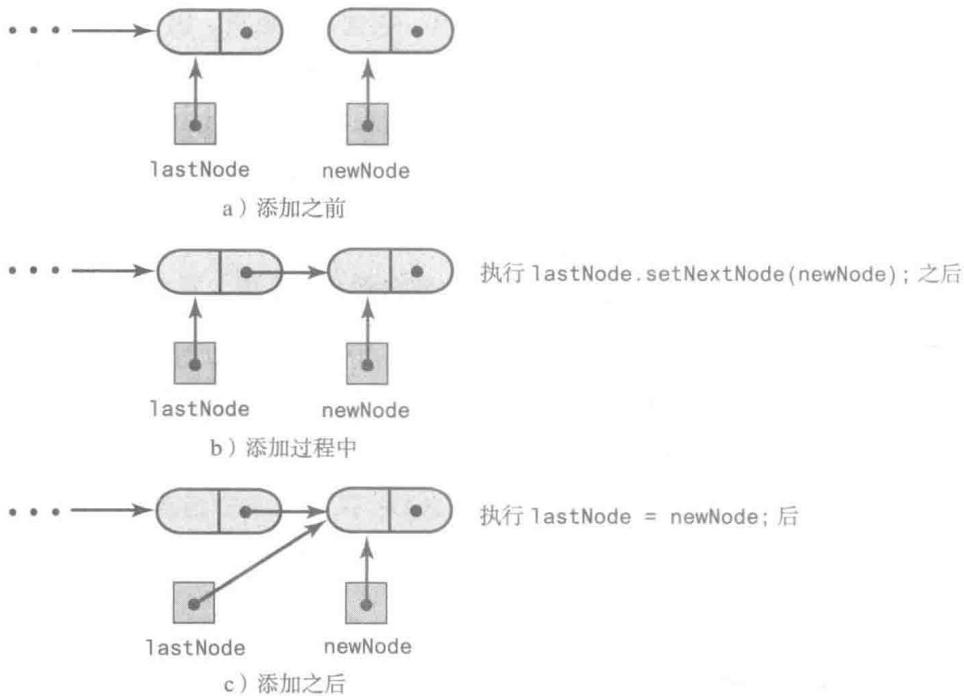


图 8-3 将新结点添加到有尾引用的非空链表的尾部

故 enqueue 的定义如下所示：

```
public void enqueue(T newEntry)
{
 Node newNode = new Node(newEntry, null);
 if (isEmpty())
 firstNode = newNode;
 else
 lastNode.setNextNode(newNode);
 lastNode = newNode;
} // end enqueue
```

这个操作不需要进行查找，并且与队列中的其他项无关。所以它的性能是  $O(1)$  的。

**8.4 获取队头项。**通过访问链表中第一个结点的数据部分，可以得到队头项。与 enqueue 一样，getFront 也是  $O(1)$  的操作。

```
public T getFront()
{
 if (isEmpty())
 throw new EmptyQueueException();
 else
 return firstNode.getData();
} // end getFront
```

**8.5 删除队头项。**方法 dequeue 获取队头项，然后通过将 firstNode 指向链表的第二个结点，从而删除链表的首结点，如图 8-4 所示。如果链表中仅含有一个结点，则 dequeue 将 firstNode 和 lastNode 都置为 null，从而使得链表为空，如图 8-5 所示。

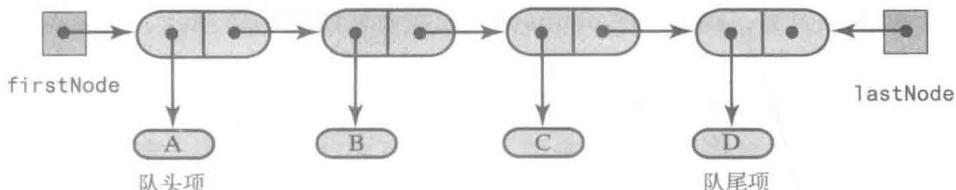
```
public T dequeue()
{
 T front = getFront(); // Might throw EmptyQueueException
 // Assertion: firstNode != null
 firstNode.setData(null);
```

```

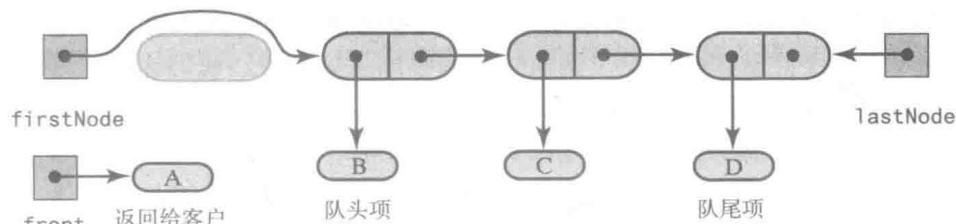
firstNode = firstNode.getNextNode();
if (firstNode == null)
 lastNode = null;
return front;
} // end dequeue
}

```

与 enqueue一样, dequeue不需要进行查找,且与队列中的其他项无关。所以其性能也是 $O(1)$ 的。

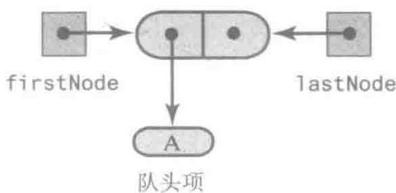


a) 含有多个项的队列

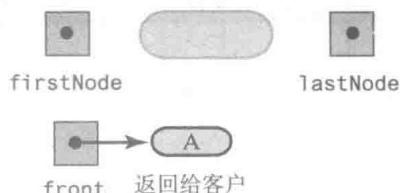


b) 删除队头项之后

图 8-4 从含有多个项的队列的队头删除一项之前和之后



a) 仅含有一个项的队列



b) 删除唯一的项之后

图 8-5 从队列中删除仅有的一项之前和之后

类中的其他方法。剩下的公有方法 isEmpty 和 clear 很简单:

8.6

```

public boolean isEmpty()
{
 return (firstNode == null) && (lastNode == null);
} // end isEmpty
public void clear()
{
 firstNode = null;
 lastNode = null;
} // end clear
}

```

**学习问题 2** 当使用结点链表实现队列时,为什么要有尾引用?



## 基于数组实现队列

8.7

如果使用数组 `queue` 保存队列中的项，则可让 `queue[0]` 是队头，如图 8-6a 所示。其中，`frontIndex` 和 `backIndex` 分别是队头项和队尾项的下标。但当删除队头项时会发生什么？如果我们坚持让新的队头项一定要在 `queue[0]` 中，则必须将每个数组项向数组头的方向移动一个位置。这样的安排使得 `dequeue` 操作的效率不高。

换一种方式，当删除队头项时，可以将其余的数组项留在当前位置。例如，如果从图 8-6a 所示的数组开始，执行两次 `dequeue`，则数组将如图 8-6b 所示。不移动数组项令人满意，但几次添加和删除后，数组可能如图 8-6c 所示的那样。队列项已经移到了数组的尾端。最后一个可用的数组元素分配给最后添加进队列的项。我们可以扩展数组，但队列中只有 3 个项。因为数组的大多数空间都未用，为什么不将这些空间用于未来的添加呢？事实上，这正是我们接下来要做的。

### 循环数组

8.8

一旦队列到达数组尾，如图 8-6c 所示，我们可以将随后进入队列的项添加到数组的开头。图 8-7 显示了在队列中进行了两次这样的添加后的数组。我们让数组好似是个循环（circular）的，这样它的第一个元素接在其最后元素之后，如图 8-7 所示。为此，我们在下标上使用取模运算。具体来说，当将项添加到队列中时，将 `backIndex` 加 1 再对数组大小取模。例如，如果 `queue` 是数组名，则使用下面语句将 `backIndex` 加 1：

```
backIndex = (backIndex + 1) % queue.length;
```

要删除一项，用类似的方式，将 `frontIndex` 加 1 再对数组大小取模。

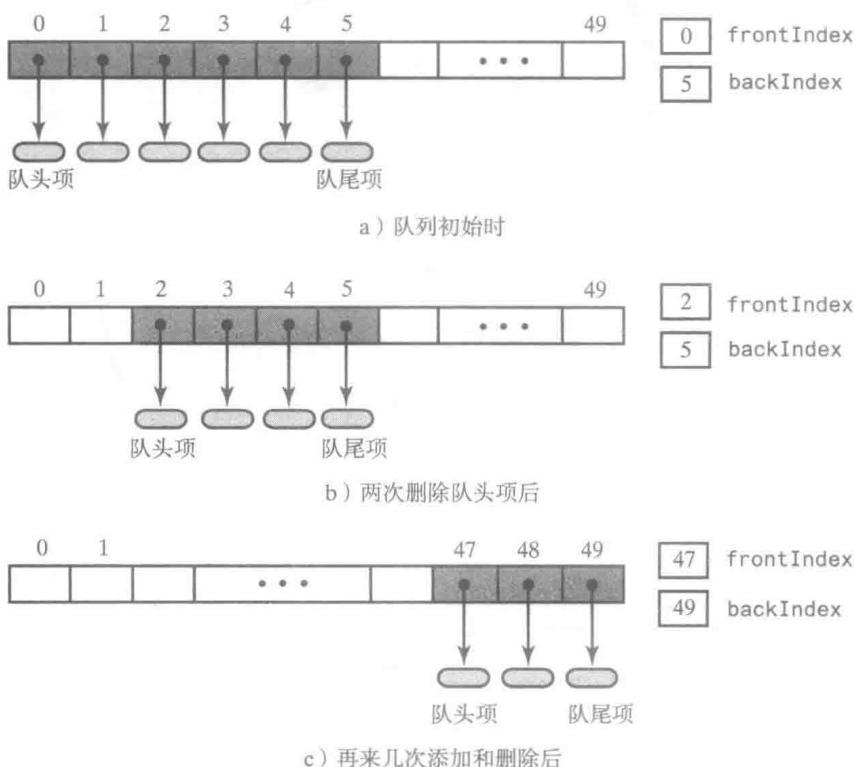


图 8-6 在添加和删除过程中，不移动任何项的表示队列的数组

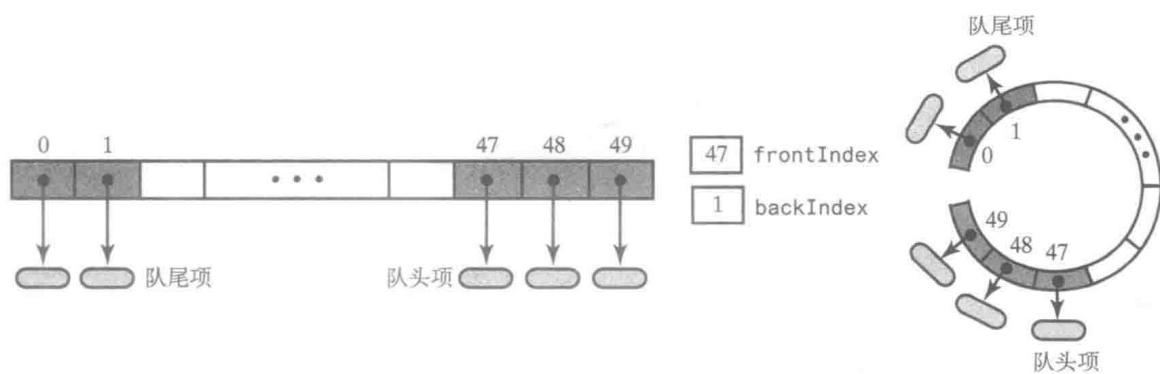
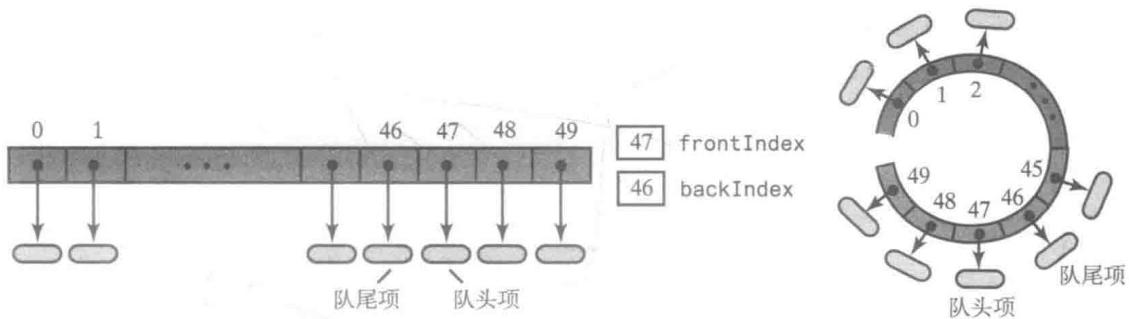


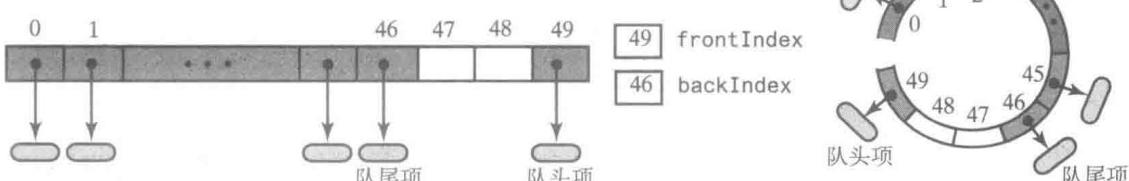
图 8-7 表示图 8-6c 中再添加两项后的队列的循环数组

**学习问题 3** 第 2 章, 当从基于数组的包中删除一项时, 我们使用数组的最后一项替代被删除的项。刚刚描述的队列的实现没有这样做。解释实现上的这个差异。

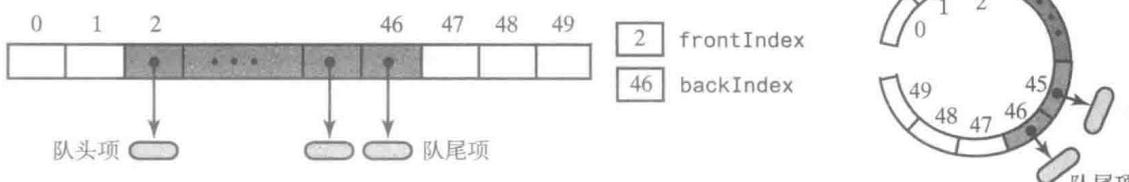
混乱。使用循环数组有时会使实现混乱。例如, 我们如何检测数组何时为满? 在图 8-7 所示的队列中添加了几个项后, 可以得到图 8-8a 所示的满队列。所以当 `frontIndex` 等于 `backIndex` + 1 时出现队满的情况。



a) 向图 8-7 所示的队列中添加更多的项直到它变满之后



b) 删除两项之后



c) 再删除三项之后

图 8-8 表示删除项后的队列的循环数组

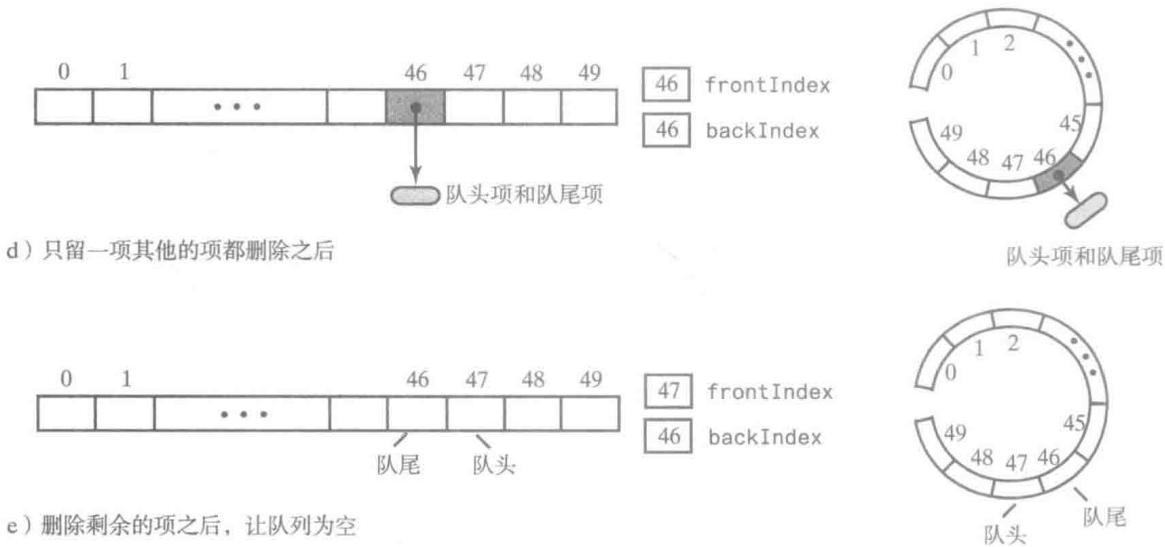


图 8-8 (续)

现在从队列中删除某些项。图 8-8b 显示执行两次 `dequeue` 后的数组。注意到，`frontIndex` 前进到 49。如果继续从队列中删除项，则 `frontIndex` 将绕回到 0 并到了那之后的位置。图 8-8c 显示再删除 3 项后的数组。当我们从队列中删除更多的项时，`frontIndex` 前进。图 8-8d 显示从队列中删除除一项外的所有项的情况。现在删除这一项。图 8-8e 中，我们看到这最后的删除使得 `frontIndex` 前进，所以它是 `backIndex+1`。虽然队列是空的，但 `frontIndex` 等于 `backIndex + 1`。这与我们在图 8-8a 中遇到的队列满时的条件是一样的。



**注：** 使用循环数组，当队列空时及满时，都满足 `frontIndex` 等于 `backIndex+1`。

正如你所见的，我们不能使用 `frontIndex` 和 `backIndex` 来测试队列是否为空或为满。一种解决办法是对队列的项数进行计数。如果计数为 0，则队列为空；如果计数等于数组的容量，则队列为满。当队列为满时，下一次的 `enqueue` 操作要在添加新项前倍增数组的大小。

让计数器作为数据域，是一种合理的实现方案，但每次 `enqueue` 和 `dequeue` 必须要更新计数。我们可以令一个数组元素不用，来避免这种额外的工作。下面将开发这个方法。

### 带一个未用元素的循环数组

8.10

让数组的一个元素不使用，则仅需检查 `frontIndex` 和 `backIndex` 的值就能让我们区分开空队列和满队列。在 Java 中，每个数组元素仅含有一个引用，所以留下一个元素不用仅浪费很少的内存。这里，我们将未用的数组元素放在队尾的后面。本章结尾处的项目 3 考虑不同的元素。

图 8-9 图示了含 7 个元素能表示最多有 6 项的队列的循环数组。当我们添加和删除项时，你应该观察对下标 `frontIndex` 和 `backIndex` 的影响。图 8-9a 显示初始时队列为空的情形。注意到，`frontIndex` 是 0，而 `backIndex` 是数组最后元素的下标。向队列中添加一项时，在 `backIndex` 的初值上加 1，所以它变为 0，如图 8-9b 所示。图 8-9c 是再进行 5 次添加后变为满的队列。现在删除队头项，并在队尾添加一项，如图 8-9d 和图 8-9e 所示。队列再次满了。重复这对操作，队列如图 8-9f 和图 8-9g 所示。现在重复删除队头项，直到队

列为空时为止。其中，完成第一次 `dequeue` 操作后的队列如图 8-9h 所示，除一项外删除所有项后的队列如图 8-9i 所示，图 8-9j 显示空队列。

概括来说，在图 8-9c、图 8-9e 和图 8-9g 中队列是满的。在上述每个例子中，如果我们将数组看作循环的，则未用元素的下标比 `backIndex` 多 1，而比 `frontIndex` 少 1。即 `frontIndex` 比 `backIndex` 多 2。所以当

`frontIndex` 等于  $(backIndex + 2) \% queue.length$

时，队列是满的。队列在图 8-9a 和图 8-9j 中是空的。这些情形中，`frontIndex` 比 `backIndex` 多 1。所以，当

`frontIndex` 等于  $(backIndex + 1) \% queue.length$

时，队列是空的。不可否认，这些准则比检查队列中的项数要复杂一些。但是，一旦使用这些准则，则其他的实现更简单且更高效，因为不需要维护计数器。

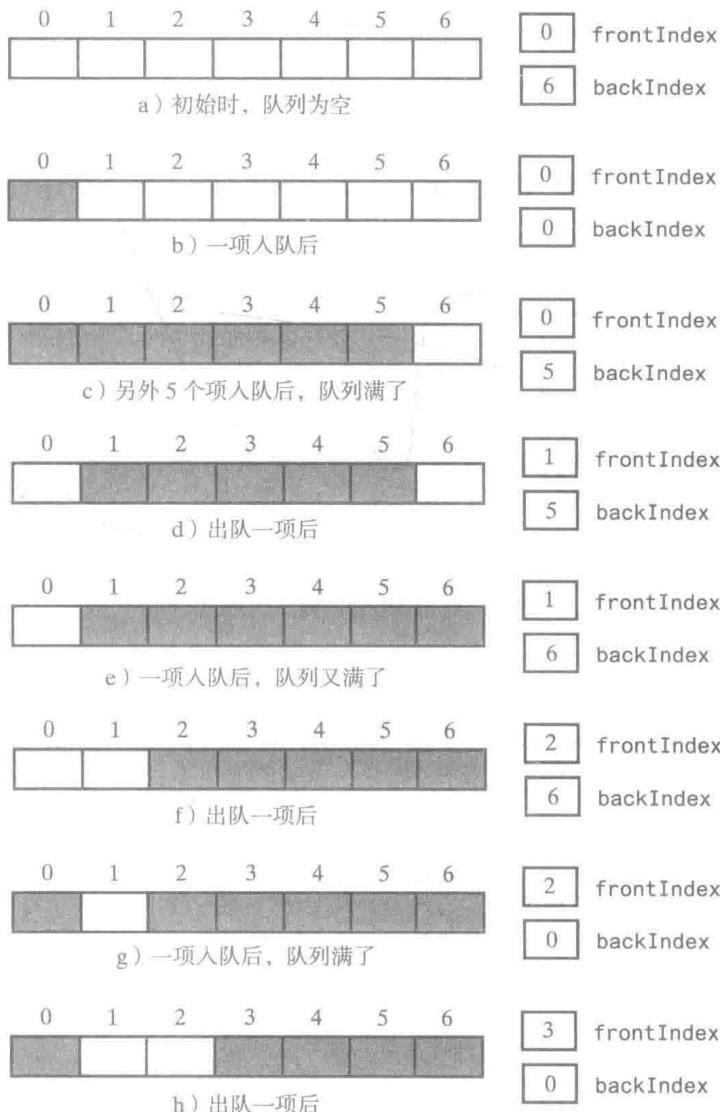


图 8-9 有 7 个元素的循环数组，含有最多 6 个项的队列



图 8-9 (续)

**8.11 类的框架。**队列基于数组的实现中，最前面是 4 个数据域和两个构造方法。数据域是队列项的数组、队头和队尾的下标、默认构造方法创建的队列的初始容量。另一个构造方法让客户选择队列的初始容量。数组的初始大小比队列的初始容量大 1。程序清单 8-2 概述了类。

### 程序清单 8-2 基于数组实现 ADT 队列的框架

```

1 /**
2 * A class that implements a queue of objects by using an array.
3 */
4 public final class ArrayQueue<T> implements QueueInterface<T>
5 {
6 private T[] queue; // Circular array of queue entries and one unused element
7 private int frontIndex;
8 private int backIndex;
9 private boolean integrityOK;
10 private static final int DEFAULT_CAPACITY = 50;
11 private static final int MAX_CAPACITY = 10000;
12
13 public ArrayQueue()
14 {
15 this(DEFAULT_CAPACITY);
16 } // end default constructor
17
18 public ArrayQueue(int initialCapacity)
19 {
20 integrityOK = false;
21 checkCapacity(initialCapacity);
22
23 // The cast is safe because the new array contains null entries
24 @SuppressWarnings("unchecked")
25 T[] tempQueue = (T[]) new Object[initialCapacity + 1];
26 queue = tempQueue;
27 frontIndex = 0;
28 backIndex = initialCapacity;
29 integrityOK = true;
30 } // end constructor
31 <Implementation of the queue operations go here. >
32 . .
33 } // end ArrayQueue

```

**8.12 在队尾添加。**方法 `enqueue` 调用私有方法 `ensureCapacity`，如果数组满了，它倍增数组，然后紧接着数组已占的最后元素的后面放置新项。要确定这个元素的下标，需要将 `backIndex` 加 1。但因为数组是循环的，所以使用运算符 %，当 `backIndex` 到达最大值时让它变为 0。

```

public void enqueue(T newEntry)
{
 checkIntegrity();
 ensureCapacity();
 backIndex = (backIndex + 1) % queue.length;
 queue[backIndex] = newEntry;
} // end enqueue

```

`ensureCapacity` 的实现不同于第 6 章给出的实现，因为这里的数组是循环的。马上就会看到如何实现它。

当不擴大数组的大小时，`enqueue` 的性能不依赖于队列中的项数。所以这种情形下它是  $O(1)$  的。但是，当数组满时，它的性能降为  $O(n)$ ，因为擴大数组是  $O(n)$  的操作。但是，如果发生这种情况，则接下来的 `enqueue` 又会是  $O(1)$  的。正如我们在段 6.9 中提到的，可以将倍增数组的开销分摊在向队列中的所有添加操作上。即我们让所有的 `enqueue` 操作共担擴大数组的开销。除非数组擴大很多次，否则每次 `enqueue` 几乎都是  $O(1)$  的。

获取队头项。方法 `getFront` 或是返回在 `frontIndex` 位置的数组项，或是如果队列为 8.13 空则抛出一个异常：

```

public T getFront()
{
 checkIntegrity();
 if (isEmpty())
 throw new EmptyQueueException();
 else
 return queue[frontIndex];
} // end getFront

```

这个操作是  $O(1)$  的。

删除队头项。与 `getFront` 一样，方法 `dequeue` 获取队头项，不同的是之后删除它。要删除图 8-10a 所示队列的队头项，可以简单地将 `frontIndex` 加 1，如图 8-10b 所示。只这一步就能满足要求，因为其他的方法有正确的处理步骤。例如，`getFront` 将返回 `queue[6]` 指向的项。不过，之前是队头且返回给客户的对象仍由数组所指向。如果实现是正确的，则这个事情也不用太在意。为安全起见，`dequeue` 方法可以在将 `frontIndex` 加 1 之前，将 `queue[frontIndex]` 设置为 `null`。这种情形下的队列如图 8-10c 所示。

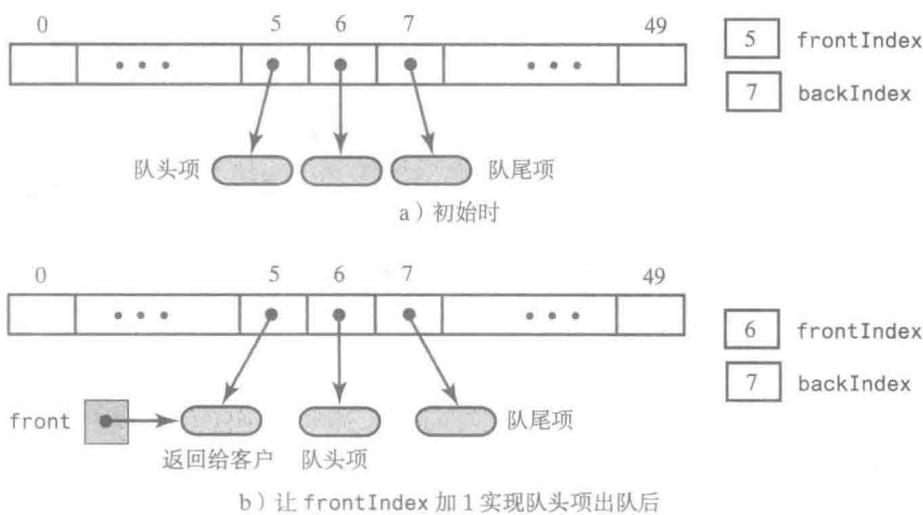
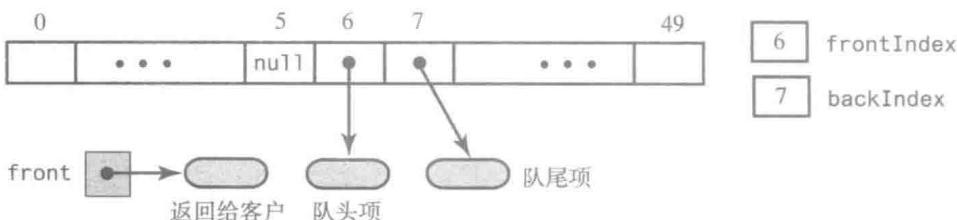


图 8-10 基于数组的队列和其删除队头项的两种方式



c) 将 frontIndex 加 1 且将 queue[frontIndex] 设置为 null 实现队头项出队后

图 8-10 (续)

前面解释的这些内容反映在 `dequeue` 的下列实现中：

```
public T dequeue()
{
 checkIntegrity();
 if (isEmpty())
 throw new EmptyQueueException();
 else
 {
 T front = queue[frontIndex];
 queue[frontIndex] = null;
 frontIndex = (frontIndex + 1) % queue.length;
 return front;
 } // end if
} // end dequeue
```

和 `getFront` 一样，`dequeue` 也是  $O(1)$  操作。

**8.15 私有方法 ensureCapacity**。正如在第 2 章段 2.35 中所见，当增大数组大小时，必须将它的项拷贝到新分配的空间中。不过我们必须要谨慎，因为这里的数组是循环数组。我们必须将项按照它们在队列中出现的次序进行拷贝。

例如，图 8-9g 中含 7 个元素的数组是满的，且再次出现在图 8-11 中。称这个数组为 `oldQueue`。分配了新的有 14 个元素的数组 `queue` 后，我们将队头从 `oldQueue[frontIndex]` 拷贝到 `queue[0]` 中。继续从原数组中将元素拷贝到新数组中，处理到原数组的尾，并绕回到它的开头，如图所示。此外，我们必须设置 `frontIndex` 和 `backIndex` 的值，以反映重新组织的数组的状态。

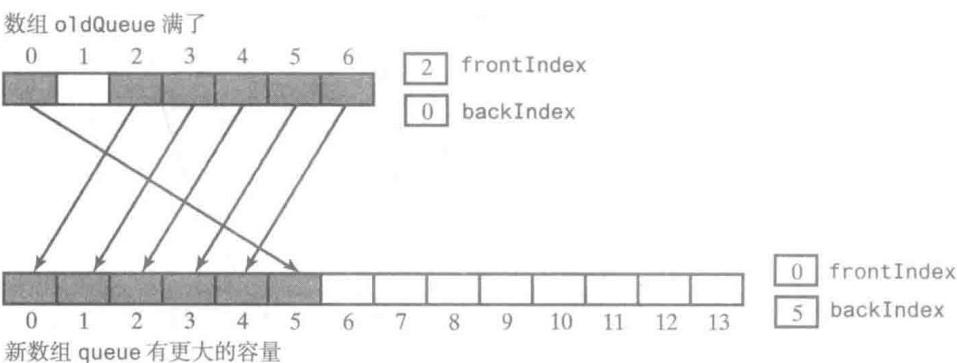


图 8-11 基于数组的队列的倍增

`ensureCapacity` 的下列定义，使用段 8.10 给出的准则，检验数组何时为满：

```
// Doubles the size of the array queue if it is full.
// Precondition: checkIntegrity has been called.
```

```

private void ensureCapacity()
{
 if (frontIndex == ((backIndex + 2) % queue.length)) // If array is full,
 { // double size of array
 T[] oldQueue = queue;
 int oldSize = oldQueue.length;
 int newSize = 2 * oldSize;
 checkCapacity(newSize - 1);
 integrityOK = false;
 // The cast is safe because the new array contains null entries
 @SuppressWarnings("unchecked")
 T[] tempQueue = (T[]) new Object[newSize];
 queue = tempQueue;
 for (int index = 0; index < oldSize - 1; index++)
 {
 queue[index] = oldQueue[frontIndex];
 frontIndex = (frontIndex + 1) % oldSize;
 } // end for
 frontIndex = 0;
 backIndex = oldSize - 2;
 integrityOK = true;
 } // end if
} // end ensureCapacity

```

你可以使用方法 `System.arraycopy` 来拷贝数组。但是，因为数组是循环的，所以这个方法必须调用两次。本章最后的练习 1 要求你按这种方式修改 `ensureCapacity`。

类的其他方法。根据我们在段 8.10 结尾处所说明的，公有方法 `isEmpty` 的实现如下：

```

public boolean isEmpty()
{
 checkIntegrity();
 return frontIndex == ((backIndex + 1) % queue.length);
} // end isEmpty

```

方法 `clear` 只需将 `frontIndex` 设置为 0，且将 `backIndex` 设置为 `queue.length - 1`。队列的其他方法会像预期的处理空队列那样来处理。但是，队列中的对象仍然保留了被分配的空间。为了释放它们，`clear` 方法应该将用于队列的每个数组元素都设置为 `null`。或者，如果 `dequeue` 方法将 `queue[frontIndex]` 设置为 `null`，则 `clear` 可以重复地调用 `dequeue`，直到队列为空时为止。我们将 `clear` 的实现留作练习。



**学习问题 4** 实现 `clear`，将用于队列的每个数组元素设置为 `null`。

**学习问题 5** 实现 `clear`，重复调用 `dequeue`，直到队列为空时为止。将这个实现与学习问题 4 中你的实现进行比较。

**学习问题 6** 如果 `queue` 是含有队列项的数组，且 `queue` 不看作循环数组，那么，将队尾放在 `queue[0]` 时的缺点是什么？



**注：**Java 之外的有些语言中，让数组元素为空会浪费空间，因为元素中含有的是一个对象而不是指向对象的引用。本章结尾的项目 4 考虑了不含未用元素且不含计数器的基于数组的队列实现。

## 队列的循环链式实现

图 8-1 显示了实现 ADT 队列的结点链表。这个链表有两个外部引用——一个指向链表

的首结点，一个指向链表的最后一个结点。回忆一下，这些引用对于队列实现是有特殊用途的，因为队列的操作影响它的两端。与你之前见过的链表一样，这个链表的最后一个结点中含有 `null`。这样的链表有时称为 **线性链表** (linear linked chain)，不管它们在头引用之外有没有尾引用。

在**循环链表** (circular linked chain) 中，最后一个结点指向第一个结点，所以哪个结点的 `next` 域中都不含有 `null` 值。尽管每个结点都指向下一个结点，但是循环链表有开始也有结尾。我们有一个外部引用指向链表的首结点，但要找到最后一个结点还必须遍历链表。通常，都会有指向首结点的引用和指向最后一个结点的引用，但不是必须都有。因为链表的最后一个结点指向其首结点，所以只用指向最后一个结点的引用，仍能快速找到首结点。图 8-12 图示了这样的一个链表。

当类使用循环链表表示队列时，它唯一的数据域是指向链表的最后结点的引用 `lastNode`。所以实现中没有维护指向首结点的引用数据域的开销。当需要这样的引用时，使用表达式 `lastNode.getNextNode()` 就可以得到它。尽管做了这些简化，这个方法不一定比本章第一节中使用的方法要好。最多也就是有所不同，当你完成本章最后的项目 5 时会明白这一点。

现在研究使用循环链表表示队列的另一种方法。

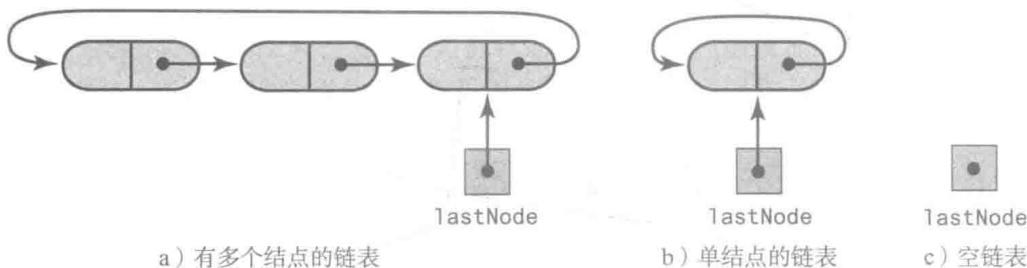


图 8-12 有指向最后结点的外部引用的循环链表

## 两部分组成的循环链表

**8.18** 当用链表——不论是线性的还是循环的——表示队列时，对队列中的每个项都对应一个结点。当向队列中添加项时，为链表分配新的结点。当从队列中删除一个项时，释放结点。

在循环数组实现中，队列用到了定长数组中可用元素中的一部分。当向队列中添加一项时，使用数组中下一个未占用的元素。当从队列中删除一项时，数组的这个元素可用于队列以后的使用。因为添加和删除都在队列的端点处进行，所以队列占用的是循环数组中连续的元素。可用元素也是连续的，这也是因为数组是循环的。所以循环数组含有两部分：一部分含有队列，另一部分可用于队列。

假定在循环链表中有两部分。组成队列的链表结点的后面是可用于队列的链表结点，如图 8-13 所示。这里，`queueNode` 指向已分配给队头的结点；`freeNode` 指向跟在队尾之后的第一个可用结点。可以把这个形态看作两个链表——一个用于队列，一个当作可用结点——它们在端点处连起来形成一个环。

可用结点没有像数组那样一下子同时分配。初始时，没有可用结点；每次向队列中添加新项时分配一个结点。但是，当从队列中删除项时，将这个结点保留在结点环中而不是释放掉它。所以，后续向队列中的添加使用来自于可用结点链表上的结点。但如果没有任何这样的可用结点了，则将分配新的结点并将它链接到链表中。

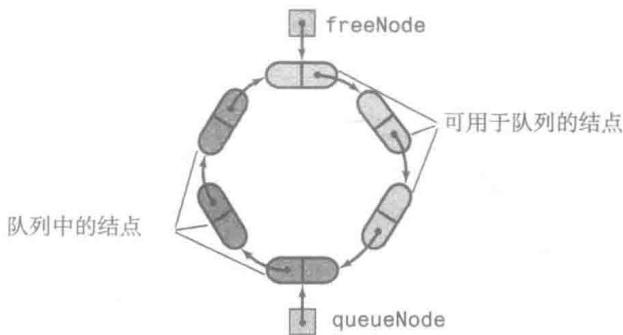


图 8-13 表示队列及可用于队列的结点的两部分组成的循环链表

如果循环链表中有一个结点未用，则很容易检测空队列或是没有可用结点。这类似于我们在段 8.10 中用到的循环数组的情况。图 8-14a 显示了空队列。`queueNode` 和 `freeNode` 都指向同一个未用结点。注意到，结点指向自己。可以说，队列是空的，因为 `queueNode` 等于 `freeNode`。

8.19

要向这个空队列中添加项时，会分配一个新结点，并将其链接到循环链表中。图 8-14b 所示为含一个项的队列的链表。为使这个图简洁，我们没有画出队列中的实际对象。虽然链表中的一个结点指向队列中的一个对象，但我们有时会说结点在队列中。

`queueNode` 指向分配给队列的结点，而 `freeNode` 仍指向未用结点。向队列中添加 3 次后，又分配了 3 个结点，并将它们链到链表中。段 8.21 会严格描述如何完成这个过程。现在链表如图 8-14c 所示。`freeNode` 仍指向未用结点。因为 `queueNode` 指向队头结点，所以获取队头项很简单。

现在，如果我们从队头删除一项，则 `queueNode` 前移，故链表如图 8-14d 所示。队头的结点并没有释放。后续的添加——因为它位于队尾——使用 `freeNode` 指向的结点。然后，将 `freeNode` 前移。图 8-14e 所示为此时的链表。注意到，这种情况下，我们并没有为所添加的项分配新的结点。

当向队列中添加时，我们如何判定是否必须分配一个新结点？如果像图 8-14e 所示的那样，`queueNode` 等于 `freeNode.getNextNode()` 时，我们就必须分配新结点。当向图 8-14d 所示的队列中添加项时不属于这种情况；因为有一个结点可用，故不需要分配新结点。但注意到，在图 8-14a 中，当队列为空时，`queueNode` 也等于 `freeNode.getNextNode()`。这样做也是合情合理的，因为要在空队列中进行添加，所以必须分配一个新结点。

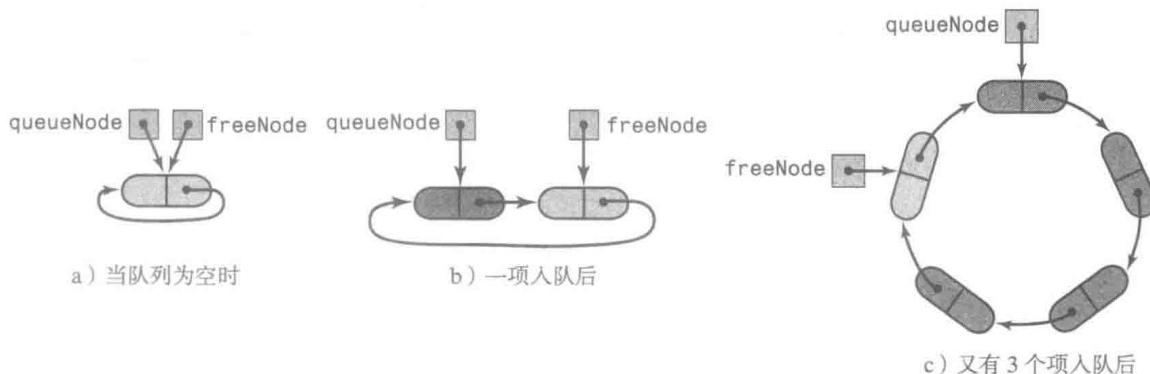


图 8-14 表示队列的两部分组成的循环链表的各种状态

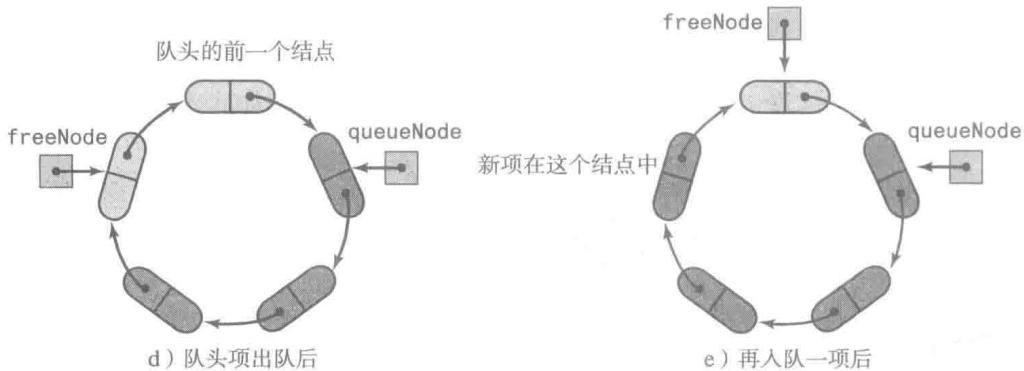


图 8-14 (续)



**注：**在实现队列的两部分组成的循环链表中，有一个结点是未用的。两个外部引用将链表分为两部分：queueNode 指向队头结点，而 freeNode 指向接在队列后的结点。如果 queueNode 等于 freeNode，则队列为空。新项可以使用 freeNode 处的结点。这个结点或是第一个可用结点，或是未用结点。如果 queueNode 等于 freeNode.getListNode()，则必须分配新的未用结点。

## 8.20

**类的框架。**使用两部分组成的循环链表实现队列的类，其数据域是 queueNode 引用和 freeNode 引用。因为链表必须至少含有一个结点，所以默认的构造方法分配一个结点，让结点指向自己，然后将 queueNode 和 freeNode 都指向这个新结点。由此，得到程序清单 8-3 中所列的类框架。

**程序清单 8-3** 使用两部分组成的循环链表实现 ADT 队列的类框架

```

1 /**
2 * A class that implements a queue of objects by using
3 * a two-part circular chain of linked nodes.
4 */
5 public final class TwoPartCircularLinkedList<T> implements QueueInterface<T>
6 {
7 private Node queueNode; // References first node in queue
8 private Node freeNode; // References node after back of queue
9
10 public TwoPartCircularLinkedList()
11 {
12 freeNode = new Node(null, null);
13 freeNode.setNextNode(freeNode);
14 queueNode = freeNode;
15 } // end default constructor
16
17 < Implementation of the queue operations go here. >
18 .
19 private class Node
20 {
21 private T data; // Queue entry
22 private Node next; // Link to next node
23
24 < Constructors and the methods getData, setData, getNextNode, and setNextNode
25 are here. >
26 .
27 } // end Node
28 } // end TwoPartCircularLinkedList

```



**程序设计技巧：**当循环链表只有一个结点时，这个结点必须指向自己。很容易忘记这一步，由此导致运行时的错误。

**在队尾添加。**向队列中添加项之前，必须看看链表中是否有可用结点。如果没有，必须分配一个新结点并将它链到链表中。将新结点插入链表中 `freeNode` 所指结点的后面，如在图 8-15a 中所做的一样。我们没有将它插入在这个结点之前，因为插入时我们需要一个指向前一结点的引用。得到这样一个引用很费时间。`freeNode` 指向的结点加入队列中，且它含有新项。新结点成为未用结点，我们让 `freeNode` 指向它，如图 8-15b 所示。

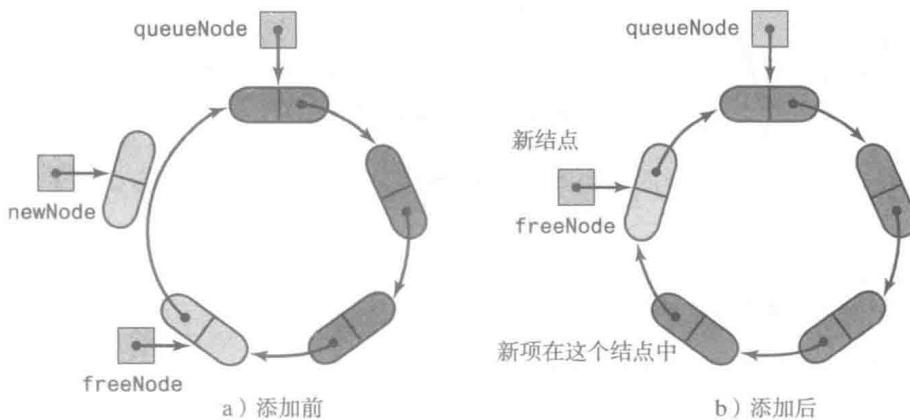


图 8-15 向队列中添加项时需要一个新结点的两部分组成的循环链表

如果链表中有一个结点可用，则使用 `freeNode` 指向的结点保存新项。图 8-16 显示了两个已分配的结点成为队列成员之前和之后的链表。每次添加后，`freeNode` 指向跟在队尾后的结点。在图 8-16b 中，这个结点可用于另一次的添加，但在图 8-16c 中，它是未用的。

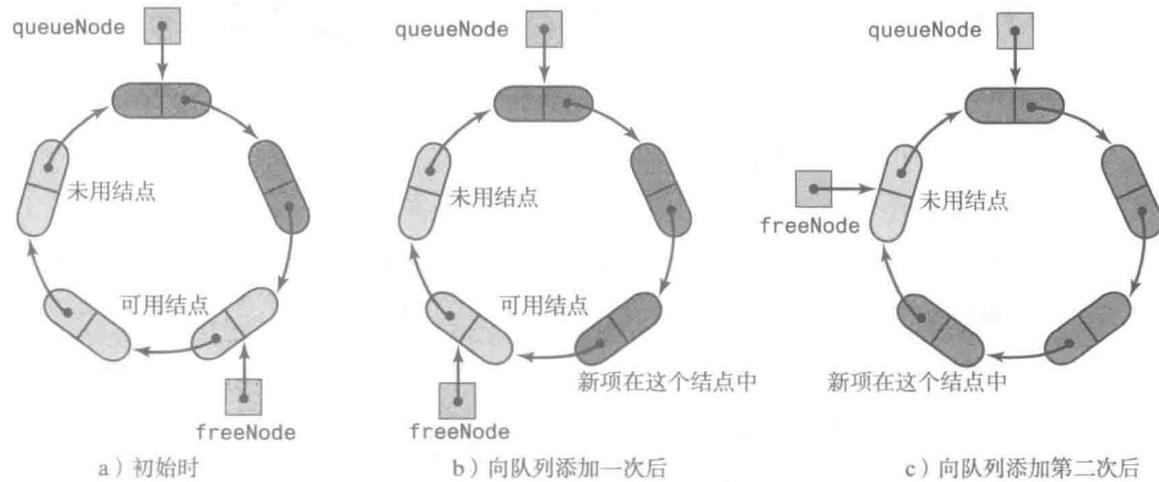


图 8-16 向队列添加时有可用结点的两部分组成的循环链表

如果我们将判断是否分配一个新结点的细节隐藏在私有方法 `isNewNodeNeeded` 中，则方法 `enqueue` 既容易写也容易理解。如果链表没有可用结点给队列使用，则它返回假。`isNewNodeNeeded` 的实现并不困难，在后面的段 8.24 中会实现它。

`enqueue` 的下列实现是  $O(1)$  的操作：

```

public void enqueue(T newEntry)
{
 freeNode.setData(newEntry);
 if (isNewNodeNeeded())
 {
 // Allocate a new node and insert it after the node that
 // freeNode references
 Node newNode = new Node(null, freeNode.getNextNode());
 freeNode.setNextNode(newNode);
 } // end if
 freeNode = freeNode.getNextNode();
} // end enqueue

```



**学习问题 7** 图 8-16c 中向队列中添加一项，需要创建一个新结点。这个新结点应该插入链表的什么位置？哪个结点应该含有这个新项？

8.22 获取队头。如果队列不空，则 queueNode 指向队头结点。所以方法 getFront 很简单：

```

public T getFront()
{
 if (isEmpty())
 throw new EmptyQueueException();
 else
 return queueNode.getData();
} // end getFront

```

这个方法是  $O(1)$  的。

8.23 删除队头。方法 dequeue 返回队头项。然后通过让 queueNode 前移，将队头结点从队列部分移到可用部分。图 8-14c 和图 8-14d 分别显示这一步骤之前和之后的情形。因为没有释放含有被删除项的结点，所以它仍然指向被删除的项。故要将结点的数据部分置为 null。

与 getFront 一样，dequeue 是  $O(1)$  的操作：

```

public T dequeue()
{
 T front = getFront(); // Might throw EmptyQueueException
 // Assertion: Queue is not empty
 queueNode.setData(null);
 queueNode = queueNode.getNextNode();
 return front;
} // end dequeue

```

8.24 类的其他方法。段 8.19 中讨论的方法 isEmpty 和 isNewNodeNeeded 如下所示。

```

public boolean isEmpty()
{
 return queueNode == freeNode;
} // end isEmpty

private boolean isNewNodeNeeded()
{
 return queueNode == freeNode.getNextNode();
} // end isNewNodeNeeded

```

注意，当链表中没有结点用于队列项时，isNewNodeNeeded 按要求返回真。这种情形下的链表如图 8-14a、图 8-14b、图 8-14c 和图 8-14e 所示。

方法 clear 中设置 queueNode 的值等于 freeNode，使队列表现为空。它保留了链表中当前的所有结点。但是，除非将这些结点的数据部分置为 null，否则队列中的对象不会被释放。将 clear 的实现留作练习。



### 学习问题 8 描述可用来实现方法 clear 的两种不同的方式。

选择链式实现。到目前为止，我们已经讨论了 ADT 队列的几种可能的链式实现。可以使用带头尾引用的线性链表，如图 8-1 所示，或等价的带一个外部引用的循环链表，如图 8-12 所示。这些实现中，从队列中删除项都会断开链表并释放链表中的结点。如果，从队列中删除项后，很少再添加项，则这样的实现很好。但如果你频繁地在删除项后又添加项，则图 8-12 所示的两部分组成的循环链表可以节省释放及再分配结点的时间。

## Java 类库：类 AbstractQueue

Java 类库的标准包 `java.util` 中含有抽象类 `AbstractQueue`。这个类实现了接口 `java.util.Queue`，且不允许队列中含有 `null` 值。回忆前一章的段 7.13，这个接口中有下列方法：

```
public boolean add(T newEntry)
public boolean offer(T newEntry)
public T remove()
public T poll()
public T element()
public T peek()
public boolean isEmpty()
public void clear()
public int size()
```

`AbstractQueue` 中，分别调用 `offer`、`poll` 和 `peek` 方法，实现了 `add`、`remove` 和 `element` 方法。

你可以继承 `AbstractQueue` 来定义队列类。你的类必须至少重写下列方法：`offer`、`poll`、`peek` 和 `size`。注意，我们在前一章提到过，`java.util.PriorityQueue` 类继承了 `AbstractQueue`，所以它实现了在接口 `java.util.Queue` 中声明的方法。

要更多了解 `AbstractQueue` 的内容，可参考 Java 类库的在线文档。

## 队列的双向链式实现

之前在段 8.1 中，我们设计了队列的链式实现，注意到，队头不应该在结点链表的结尾。如果是，我们将遍历整个链表才能得到指向下一个结点的引用，从而才能删除队头项。

虽然将队头放在链头能解决我们的问题，但对于双端队列却不行。我们必须能从双端队列的队头及队尾两端进行删除。所以即使双端队列的队头在链头，双端队列的队尾也不会在链尾——问题就出在这里。

链表中的每个结点仅指向下一个结点。所以带头引用的一个链表，允许我们从第一个结点开始，一个结点一个结点地向前移动。有尾引用可让我们访问链表中的最后一个结点，但不能访问倒数第二个结点。即我们不能从一个结点反向移动，而这正是当删除双端队列队尾时我们要做的。

我们需要的结点，除了能指向链表中的下一个结点，还应能指向链表中的前一个结点。我们称这样的结点组成的链表为双向链表（doubly linked chain）。当必须要有所区别时，有时称原来的链表为单链表（singly linked chain）。图 8-17 所示为一个带头尾引用的双向链表。内部结点既指向下一个结点也指向前一个结点，第一个结点和最后一个结点都含有一

个 null 引用。所以，当从首结点开始遍历到最后一个结点，在到达最后结点时会遇到 null 值。类似地，当从最后一个结点遍历到首结点，在到达首结点时会遇到 null 值。

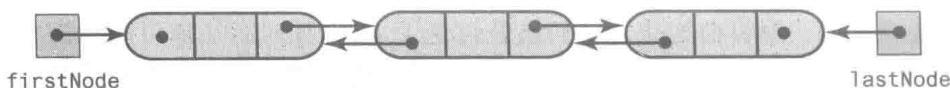


图 8-17 带头尾引用的双向链表

双向链表中的结点是类似于类 Node 的内部类的一个实例。我们称这个内部类为 DLNode，给它定义 3 个数据域：next 和 previous 指向另外两个结点，而 data 是指向结点数据的引用。DLNode 还有方法 getData、setData、getNextNode、setNextNode、getPreviousNode 和 setPreviousNode。

8.29

**类的框架。**双端队列的双向链表实现的开头部分非常类似于段 8.2 所给的队列的链式实现。类有两个数据域——firstNode 和 lastNode——默认构造方法将其都设置为 null，如你在程序清单 8-4 中所见。

#### 程序清单 8-4 ADT 双端队列的链式实现框架

```

1 /**
2 * A class that implements a deque of objects by using
3 * a chain of doubly linked nodes.
4 */
5 public final class LinkedDeque<T> implements DequeInterface<T>
6 {
7 private DLNode firstNode; // References node at front of deque
8 private DLNode lastNode; // References node at back of deque
9
10 public LinkedDeque()
11 {
12 firstNode = null;
13 lastNode = null;
14 } // end default constructor
15
16 < Implementations of the deque operations go here. >
17 .
18 private class DLNode
19 {
20 private T data; // Deque entry
21 private DLNode next; // Link to next node
22 private DLNode previous; // Link to previous node
23
24 < Constructors and the methods getData, setData, getNextNode, setNextNode,
25 getPreviousNode, and setPreviousNode are here. >
26 .
27 } // end DLNode
28 } // end LinkedDeque

```

8.30

**添加项。**方法 addToBack 的实现很像段 8.3 中所给的 enqueue 的实现。两个方法都将一个结点添加到链尾，所以链表的当前最后结点指向新结点。这里，我们还通过将双端队列的数据域 lastNode 传给结点的构造方法，从而让新结点指向当前最后结点。添加到非空链表的链尾如图 8-18 所示。方法的实现如下：

```

public void addToBack(T newEntry)
{
 DLNode newNode = new DLNode(lastNode, newEntry, null);
 if (isEmpty())

```

```

 firstNode = newNode;
 else
 lastNode.setNextNode(newNode);
 lastNode = newNode;
 } // end addToBack
}

```

除了名字之外，方法不同于 enqueue 的地方仅在于分配新结点的语句。注意到，DLNode 的构造方法的参数依次是 previousNode、nodeData 和 nextNode。

方法 addToFront 的实现类似。当向双向链表的链头添加时，必须将双端队列的数据域 firstNode 传给结点的构造方法，从而让链表的当前首结点指向新结点。图 8-19 图示了新结点添加到非空链表的链头的情形。将下列 addToFront 的定义，与刚刚给出的 addToBack 的定义进行比较：

```

public void addToFront(T newEntry)
{
 DLNode newNode = new DLNode(null, newEntry, firstNode);

 if (isEmpty())
 lastNode = newNode;
 else
 firstNode.setPreviousNode(newNode);

 firstNode = newNode;
} // end addToFront

```

如上所示，addToFront 和 addToBack 都是  $O(1)$  的操作。

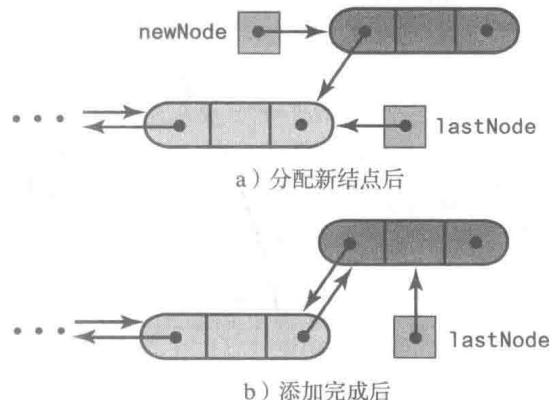


图 8-18 在非空双端队列的队尾添加

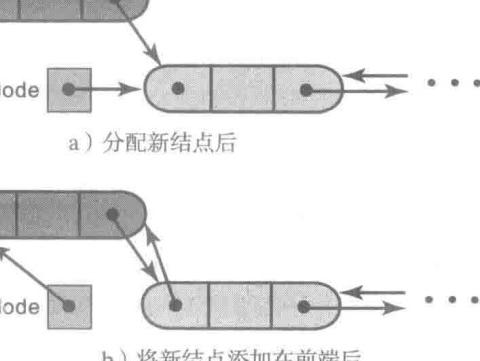


图 8-19 在非空双端队列的前端添加

**删除项。**方法 removeFront 的实现非常类似于段 8.5 给出的 dequeue，但它还要再执行另外的一步。在分离出首结点后，如果双端队列不空，则 removeFront 必须将新的首结点中的 previous 数据域置为 null。这个步骤写在下面代码的 else 子句中：

```

public T removeFront()
{
 T front = getFront(); // Might throw EmptyQueueException
}

```

```

// Assertion: firstNode != null
firstNode = firstNode.getNextNode();
if (firstNode == null)
 lastNode = null;
else
 firstNode.setPreviousNode(null);

return front;
} // end removeFront

```

除了名字外，这个方法与 `dequeue` 的不同之处仅在于多了一个 `else` 子句。图 8-20 展示了对含有至少两项的双端队列执行 `removeFront` 的效果。

方法 `removeBack` 的实现类似：

```

public T removeBack()
{
 T back = getBack(); // Might throw EmptyQueueException;
 // Assertion: lastNode != null
 lastNode = lastNode.getPreviousNode();

 if (lastNode == null)
 firstNode = null;
 else
 lastNode.setNextNode(null);
 return back;
} // end removeBack

```

`removeFront` 和 `removeBack` 的实现都是  $O(1)$  的。

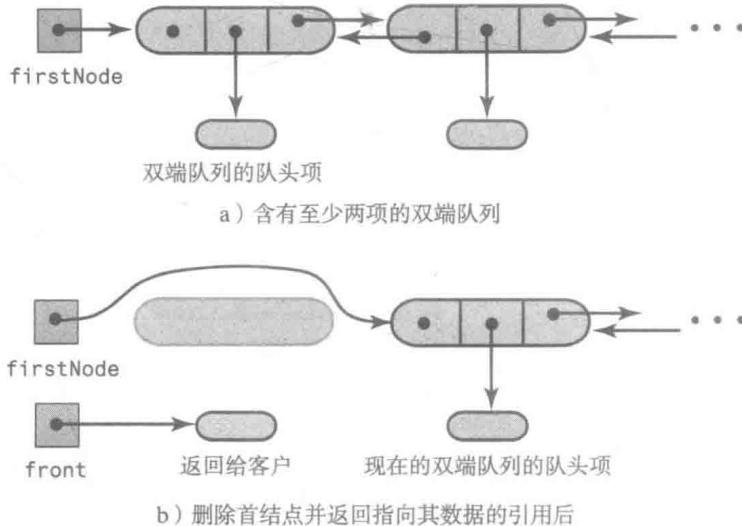


图 8-20 从含有至少两项的双端队列中删除队头项

8.32 获取项。方法 `getFront` 的实现与段 8.4 中为队列实现的方法相同。方法 `getBack` 与 `getFront` 的实现类似，留作练习。`getFront` 和 `getBack` 都是  $O(1)$  的操作。



学习问题 9 当用双向链表保存双端队列的项时，为 ADT 双端队列实现 `getBack` 方法。

8.33

重用这个实现。一旦实现了 ADT 双端队列，就可以用它来实现其他的 ADT，例如队列和栈。这些实现都非常简单，留作练习。

 注：在双向链表中，首结点和最后一个结点都含有一个 null 引用，因为首结点没有前一个结点，而最后一个结点的后面也没有结点。在循环双向链表（circular doubly linked chain）中，首结点指向最后结点，最后结点指向首结点。只需要一个外部引用——指向首结点的引用——因为你可以从首结点快速得到最后一个结点。可以使用循环双向链表来实现 ADT 双端队列。项目 9 要求你完成这个任务。

## 优先队列的可能实现方案

可以使用数组、链表或是向量来实现 ADT 优先队列。每种情形下，都要按项的优先级维护项的有序性。对数组来说，有最高优先级的项应该位于数组的结尾，所以删除它后其他项仍在原地不动。图 8-21a 图示了这个实现方案。8.34

如果优先队列的项保存在链表中，则有最高优先级的项应该位于链表的开头，这是最容易删除的位置。图 8-21b 展示了这样的一个链表。

第 10 章将介绍 ADT 线性表，第 17 章将讨论称为有序表的一种线性表。有序表可以按优先级次序维护优先队列中的项，可为我们做很多事情。第 17 章结尾的项目 10 将要求你完成这个实现。

第 24 章描述了使用称为堆的 ADT 来更高效的实现优先队列的方法。

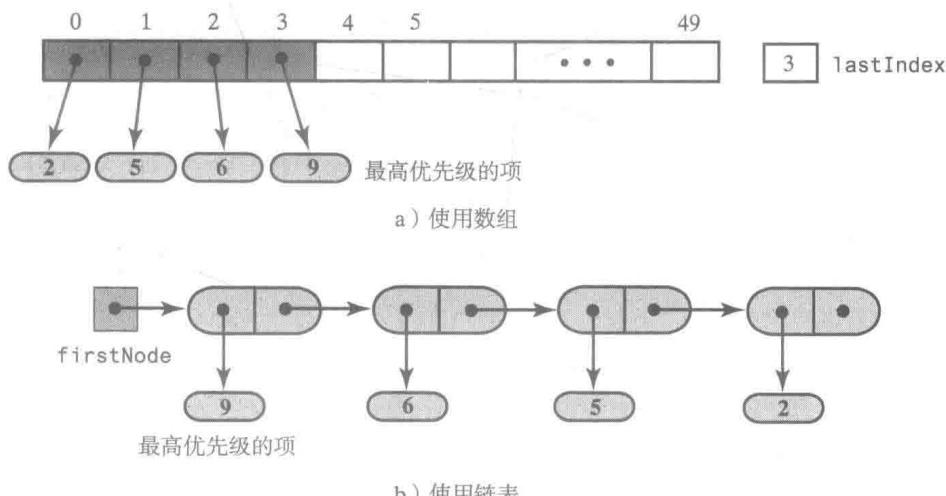


图 8-21 优先队列的两种可能的实现方案

## 本章小结

- 可以使用带头引用和尾引用的结点链表实现队列。链表中的首结点表示队头，因为删除或访问链表的首结点比其他结点更快。尾引用能让你快速将结点添加到链尾，即队尾。
- 链式实现的队列操作都是  $O(1)$  的。
- 可以使用数组实现队列。队列项一旦添加到数组中，它就不再移动。多次添加后，将用到了数组的最后元素。但是删除将使数组的开头元素空闲出来。所以即使数组没满，但看起来是满了。为解决这个问题，将数组看作循环的。

- 基于数组实现的队列操作都是  $O(1)$  的。但是，当数组满时，enqueue 需要倍增数组。这种情况下，enqueue 是  $O(n)$  的。一般的，将扩大数组的开销分摊在队列的所有添加操作上。如果数组偶尔才会扩大，则每次 enqueue 操作仍几乎是  $O(1)$  的。
- 循环链表中，每个结点指向链表中的下一个结点。所以哪个结点的 next 域中都不含有 null 值。循环链表可以有开始点和结束点。因为链表中的最后结点指向首结点，所以指向最后结点的外部引用就可以方便地访问链表的最后结点和它的首结点。
- 可以使用循环链表实现队列，与使用带头尾引用的线性链表的实现方式非常类似。两种链表中，dequeue 都删除结点并释放它。
- 队列的另一种实现方式是使用两部分组成的循环链表。一部分用于队列，另一部分含有一个未用结点及所有可用结点。这种实现中，dequeue 从队列中删除项，但并不从链表中删除结点。而是将结点加入链表的可用结点部分。
- 因为双端队列在其两端进行添加和删除操作，所以可以使用双向链表，其结点中有指向下一结点及前一个结点的引用。用带头尾引用的双向链表实现双端队列时，能提供  $O(1)$  的操作。
- 在循环双向链表中，每个结点指向链表中的下一个结点，还指向前一个结点。哪个结点的 next 域和 previous 域都不含有 null 值。指向首结点的外部引用可以快速访问链表的最后结点和它的首结点。可以使用循环双向链表实现双端队列。
- 可以使用数组或是链表实现优先队列，但更高效的实现是使用堆。第 24 章将介绍 ADT 堆。

## 程序设计技巧

- 当循环链表中只有一个结点时，这个结点必须指向自己。很容易忘记这一步，由此导致运行时的错误。

## 练习

1. 段 8.15 在使用数组实现 ADT 队列时定义了私有方法 ensureCapacity。修改这个方法，使用 `System.arraycopy` 将原数组的内容拷贝到新扩展的数组中。
2. 段 8.24 描述了使用两部分组成的循环链表表示队列时方法 clear 的实现。写出 clear 的两种不同的实现方法。一个版本应该重复调用 dequeue。另一个版本应该将队列中每个结点的数据部分置为 null。
3. 假定想在队列的类中添加一个方法，将两个队列链接在一起。这个方法将第二个队列中的所有项添加在第一个队列的最后。方法头如下：

```
public void splice(QueueInterface<T> anotherQueue)
```

用这种方式实现这个方法，让它在实现了 QueueInterface<T> 的任何类中都能正确使用。

4. 考虑练习 3 中描述的方法 splice。为类 `ArrayQueue` 实现这个方法。利用使用数组表示队列的操作效能。
5. 考虑练习 3 中描述的方法 splice。为类 `LinkedQueue` 实现这个方法。利用使用链表表示队列的操作效能。
6. 使用大 O 符号，描述类 `ArrayQueue` 中每个队列操作的时间复杂度。简要解释你的答案。
7. 使用大 O 符号，描述类 `LinkedDeque` 中每个双端队列操作的时间复杂度。简要解释你的答案。

8. 使用 ADT 双端队列保存项，实现 ADT 队列。
9. 使用 ADT 双端队列保存项，实现 ADT 栈。
10. 描述使用两个栈实现队列的思想，说明它的效率。
11. 使用向量保存项，实现 ADT 双端队列。
12. 考虑使用优先队列的一个应用。你有两种可用的实现方案。一个实现方案是使用数组来维护优先队列中的项，而另一个是使用链表。对下列施加于优先队列的每个操作序列，比较这两种实现的性能。
  - a. 插入有优先级  $1, 2, 3, \dots, 99, 100$  的 100 个对象。
  - b. 插入有优先级  $100, 99, 98, \dots, 2, 1$  的 100 个对象。
  - c. 添加有  $1 \sim 100$  之间随机优先级的 100 个对象。
  - d. 从含 100 个有  $1 \sim 100$  之间优先级的对象的优先队列开始，删除所有的对象。
  - e. 从含 100 个有  $1 \sim 100$  之间优先级的对象的优先队列开始，重复进行下列操作对 1000 次：
    - 添加具有  $1 \sim 100$  之间随机优先级的一个项。
    - 删除一个项。

## 项目

1. 使用标准类 `Vector` 的一个实例实现队列。将这个实现与本章给出的基于数组的实现方法相比较，结果如何？
2. 使用段 8.8 和段 8.9 描述的循环数组实现队列。对项进行计数，以确定队列是空或是满。
3. 段 8.10 中介绍的 ADT 队列的实现，使用了一个未用元素的循环数组。修改这个实现，让未用元素总位于队头的前面，`frontIndex` 是这个未用元素的下标。让 `backIndex` 是队尾项的下标。初始时，`frontIndex` 和 `backIndex` 都置为队列容量的最大值（数组容量总比这个值大 1）。你可以通过检查这些下标来区分空队列与满队列。为此要执行哪些测试？
4. 本章中使用数组实现的 ADT 队列用到了一个循环数组。一种实现是对队列中的项进行计数，而另一种实现是在数组中留一个未用元素。我们使用这些策略来辨别何时队列为空，何时队列为满。

还可能有第三种策略。它在循环数组中不进行计数，也没有一个未用元素。将 `frontIndex` 初始化为 0，将 `backIndex` 初始化为 -1 后，当对这两个域加 1 时不使用取模运算。而是在计算数组下标时执行取模运算，但不改变 `frontIndex` 和 `backIndex` 的值。所以，如果 `queue` 是数组，则 `queue[frontIndex % queue.length]` 是队头项，队尾项是 `queue[backIndex % queue.length]`。

现在，如果 `backIndex` 小于 `frontIndex`，则队列为空。队列中项的个数是 `backIndex - frontIndex + 1`。可以将这个数与数组大小进行比较，来查看数组是否已满。

因为 `frontIndex` 和 `backIndex` 可以持续增大，有可能变得太大了而不能表示。为了降低这种情况发生的可能性，实现中，每当检测到空队列时，就将 `frontIndex` 设置为 0，将 `backIndex` 设置为 -1。注意到，向满队列中添加项时，会调用 `ensureCapacity`，它将 `frontIndex` 设置为 0，而将 `backIndex` 设置为队尾项的下标。

完成这个基于数组的 ADT 队列的实现。

5. 使用循环链表实现 ADT 队列，如图 8-12 所示。回忆一下，这个链表仅有一个外部引用指向它的最后一个结点。
6. 考虑一种队列，一个对象只允许拷贝到队列中一次。如果向队列中添加一个对象，但它已经在队列中，则队列保持不变。这个队列有另一个操作 `moveToBack`，如果在队列中找到对象，将它移到队尾。如果对象不在队列中，则将它添加到队尾。

创建接口 `NoDuplicatesQueueInterface`，它派生于 `QueueInterface`。然后给出基于

数组的 `NoDuplicatesQueueInterface` 的实现。最后，写一个程序，充分论证这个新类。

7. 使用数组保存项，实现 ADT 双端队列。需要时动态扩展数组。
8. 实现段 8.28 中所描述的双向链表时的一个困难是，在链头及链尾操作的几种特殊情况。如果链表永远不空则可以排除这些情形。所以每个链表开始时都带一个不用于数据的哑结点 (dummy node)。使用哑结点修改本章给出的双端队列的实现。
9. 使用循环双向链表（见段 8.33 结尾处的注）实现 ADT 双端队列。
10. 重复前一个项目，但在链表中增加一个哑结点，如项目 8 所述。
11. 在项目 6 中，创建了一个不允许有重复值的队列。本项目中，将创建一个不允许有重复值的双端队列。双端队列操作 `addToBack` 和 `addToFront` 的功能应该类似于项目 6 中修改后的 `enqueue` 方法。增加两个操作 `moveToBack` 和 `moveToFront`。

创建接口 `NoDuplicatesDequeInterface`，它派生于 `DequeInterface`。然后写 `NoDuplicatesDequeInterface` 的链式实现。最后写一个程序充分论证这个新类。

12. 使用数组实现 ADT 优先队列，如图 8-21a 所示。
13. 使用结点链表实现 ADT 优先队列，如图 8-21b 所示。
14. 修改前一章段 7.19 中给出的 ADT 优先队列的接口，使用下列方法替换 `add` 方法：

```
public void add(T newEntry, Comparable<? super T> priorityValue)
```

客户为这个方法提供项及其优先级的值。优先队列不使用 `newEntry` 的 `compareTo` 方法来确定它的优先级。实现优先队列的这个版本。

15. 在项目 6 中创建了一个不允许有重复值的队列。本项目将创建一个不允许有重复值的优先队列。`add` 操作的功能应该类似于项目 6 中修改后的 `enqueue` 方法。本例中，用于相等的测试不应该包含优先级，所以 `add` 方法的方法头应该修改为前一个项目中所给定的那样。新的操作 `move` 将修改给定项的优先级，如果它已经在优先队列中。如果项不在优先队列中，则 `move` 将用所给的优先级来添加它。

创建用于不允许有重复值的优先队列的接口。然后写一个实现这个接口的类。最后写一个程序充分论证这个新类。

16. 实现队列的优先队列，如第 7 章项目 7 所描述的。
17. ADT 随机队列 (randomized queue) 类似于一个队列，但删除及取值操作的项是随机选择的而不是处理队头项。如果它们遇到一个空的随机队列，则这些操作应该返回 `null`。
  - a. 写一个 Java 接口，为随机队列规范说明其方法。取值操作命名为 `get` 而不是 `getFront`。
  - b. 定义随机队列类，命名为 `RandomizedQueue`，它实现 a 中创建的接口。

## 递 归

先修章节：附录B、第2章、第3章、第4章、第5章

### 目标

学习完本章后，应该能够

- 判定所给的递归方法是否能在有限时间内顺利结束
- 写一个递归方法
- 评估递归方法的时间效率
- 识别尾递归并能用迭代来替代

重复是很多算法的主要特征。事实上，快速的重复动作是计算机的主要能力。有两类问题求解过程涉及重复，它们可称为迭代和递归。事实上，大多数程序设计语言都提供迭代和递归这两种重复结构。

你了解迭代，因为你知道如何写一个循环。不管你使用哪种循环结构——`for`、`while`或`do`，循环中都包含想要重复执行的语句及控制重复次数的机制。循环可能是一个计数循环，例如计数为1,2,3,4,5或是5,4,3,2,1时重复；也可能是当布尔变量或表达式为真时重复执行。迭代常常能提供直接及高效的方法去实现重复过程。

有时，迭代方案会令人费解或非常复杂。对某些问题，找到或验证这样的方案不是件简单的任务。这些情形下，递归可以提供优雅的替代方案。有些递归方案可能是最优的选择，有些能有助于找到更好的迭代方案，有些则完全不能用，因为它们的效率极低。但是递归仍然是重要的问题求解策略，特别是在加密和图像处理领域。

本章将介绍如何递归地思考问题。

## 什么是递归

你可以雇一位承包商建一所房子，承包商又雇了几个分包商完成房子的各个部分，每个分包商可能再雇其他的分包商来帮忙。当你解决一个问题时也可以使用相同的方法，即将问题分解为更小的问题，并写出解决这些问题的方法。问题求解过程中每次具体的变形，除了其大小外，较小的问题与原来的问题是一样的。这个特殊的过程称为递归（recursion）。

假定你能通过解决相同但是较小的问题来求解一个问题。如何来求解这个较小的问题呢？如果再次使用递归，则必须解决与原始问题都一样而只是规模更小的问题。如何用可能达成求解目标的另一个问题来替代这个问题呢？递归成功的一个关键是，最终你能到达一个较小的问题，而这个较小问题的解决方案你是知道的，或是因为答案很明显，或是因为已经给出了答案。这个最小问题的求解或许不是原始问题的求解方案，但它能帮助你达成目标。

无论是求解更小问题之前或之后，通常你都解决了问题的一部分。这个部分与其他的更小部分的解决方案一起，得到更大问题的求解方案。

让我们来看一个例子。

9.2



**示例：倒计时。**现在是新年除夕夜，巨大的气球落在时代广场。人群倒数最后 10 秒：“10,9,8,⋯⋯”。假定我要求你从某个正整数比如 10 开始倒数到 1。你可以喊出“10”，然后让一位朋友从 9 开始倒数。从 9 开始倒数，除了要做的事情少一点之外，这是与从 10 开始倒数完全一样的问题。这是一个更小的问题。

要从 9 开始倒数，你的朋友喊出“9”，然后让一位朋友从 8 开始倒数。这个事件序列直到最终要求某个人的朋友从 1 开始倒数。这个朋友只简单地喊“1”。不再需要其他的朋友。这些事件如图 9-1 所示。

在这个示例中，我让你完成一件任务。你明白，你可以完成部分工作并要求朋友完成其余的任务。你知道，你朋友的任务和原始任务是一样的，只是更小而已。你还知道，当你朋友完成这个更小的任务时，你的任务也将完成。刚刚描述的过程中没有提到的是每位朋友在完成任务时给前一个人的信号。



图 9-1 从 10 开始倒数

为了能提供这个信号，当你从 10 开始倒数时，我需要你当做完时要告诉我。我不关心如何或是谁来做这件事，只要完成时告诉我就可以了。我可以打个盹儿，直到我听到你的声音。类似地，当你要求一位朋友从 9 开始倒数时，你也不必关心你的朋友如何完成这个任务。你只需要知道何时完成，这样你可以告诉我你已经完成了。你在等待时也可以打个盹儿。



注：递归是将一个问题划分成同样的但更小的问题的求解过程。

最终，我们有一组打盹儿的人在等待某人说“我做完了”。第一个说这话的人是喊“1”的那个人，如图 9-1 所示，因为那个人不需要帮助就能从 1 开始倒数。本例中，在那个时刻问题已经解决，但我不知道，因为我仍在睡觉。喊“1”的人向喊“2”的人说“我做完了”。喊“2”的人醒了，并且向喊“3”的人说“我做完了”，以此类推，直到你说“我做完了”。任务完成，感谢你的帮助，我不知道你如何完成的，而且我不需要知道！

这与 Java 有什么关系呢？在前一个例子中，你扮演了一个 Java 方法。我（客户）要求你（递归方法）从 10 开始倒数。当你请求朋友的帮助时，你调用一个从 9 开始倒数的方法。但你不是调用其他的方法，你调用的是你自己！

 注：调用自己的方法称为递归方法（recursive method）。调用是递归调用（recursive call 或 recursive invocation）。

下列 Java 方法从一个给定的正整数开始倒数，每行显示一个整数。

```
/** Counts down from a given positive integer.
 * @param integer An integer > 0. */
public static void countDown(int integer)
{
 System.out.println(integer);
 if (integer > 1)
 countDown(integer - 1);
} // end countDown
```

因为给定的整数是正的，故方法可以立即显示它。这一步类似于前一个例子中你喊“10”的过程。接下来，方法问你是否完成。如果所给的整数是 1，则不用再做其他事情了。但如果所给的整数大于 1，则必须从 `integer - 1` 开始倒数。我们已经注意到，这个任务更小，但除此之外，它与原始问题是一样的。我们如何来求解这个新问题呢？我们调用一个方法，而 `countDown` 就是这样一个方法。此时我们还没写完它，不过这不要紧。

方法 `countDown` 真能起作用吗？我们马上将跟踪 `countDown` 的执行过程，使你明白它能工作，也向你展示它是如何工作的。但是跟踪递归方法有些复杂，通常你不必跟踪它们。当你写递归方法时，如果遵循了一定的准则，则可以保证它是能工作的。

设计一个递归方案时，必须回答某些问题。



注：当设计一个递归方案时要回答的问题

- 方案哪个部分的工作能让你直接完成？
- 哪些较小且相同的问题已有了求解方案，当加上你的贡献时，能提供对原问题的求解？
- 过程何时结束？即哪个更小但相同的问题已有能让你达成目标或基础情形（base case）的已知的解决方案？

对于 `countDown` 方法，对这些问题的回答如下：

- 方法 `countDown` 显示所给整数，这个作为解的一部分，可以直接完成。本例中这部分恰好是最先出现的，但不总是最先出现。
- 更小的问题是从 `integer - 1` 开始倒数。当方法递归调用自己时它求解更小的问题。

9.3

9.4

- if 语句询问过程是否到达了基础情形。此处，当 integer 是 1 时出现基础情形。因为方法在检查 integer 之前已经显示了它，故一旦确认是基础情形，就什么也不用再做。



### 注：成功递归的设计原则

要写一个正确执行的递归方法，一般应该遵守下列设计原则：

- 必须给方法一个输入值，通常作为参数给出。
- 方法定义中必须含有使用了该输入值并能导向不同情形的逻辑。一般这样的逻辑包含一个 if 语句或一个 switch 语句。
- 这些情形中的一个或多个应该提供了不再需要递归的解决方案。这些是基础情形，或终止情形 (stopping case)。
- 一个或多个情形中必须包含对方法的递归调用。这些递归调用中应该含有一些步骤，通过使用“更小”的参数，或者说由方法完成的“更小”版本的任务的求解，在某种意义上逐步导向基础情形。



### 程序设计技巧：无穷递归

不检查基础情形，或缺少基础情形的递归方法，将“永远”执行。这种情形称为无穷递归 (infinite recursion)。

**9.5** 在跟踪方法 countDown 之前，应注意可以有几种不同的方法写它的代码。例如，这个方法的初稿可能是下面这个样子的：

```
public static void countDown(int integer)
{
 if (integer == 1)
 System.out.println(integer);
 else
 {
 System.out.println(integer);
 countDown(integer - 1);
 } // end if
} // end countDown
```

这里，程序员先考虑基础情形。方案清楚且完全可接受，不过你或许会想避免在两个情形中都出现的冗余的 println 语句。

**9.6** 删去刚提到的冗余，可能得到段 9.3 中所给的版本，或如下这种写法：

```
public static void countDown(int integer)
{
 if (integer >= 1)
 {
 System.out.println(integer);
 countDown(integer - 1);
 } // end if
} // end countDown
```

当 integer 是 1 时，这个方法将产生递归调用 countDown(0)。结果发现，到达了这个方法的基础情形，且什么也不显示。

所有这 3 个版本的 countDown 都能得到正确结果，可能还有其他的版本。选择对你来说最清楚的一个。

**9.7** 我们用刚在段 9.6 中给出的 countDown 与下面这个迭代版本进行比较：

```
// Iterative version.
public static void countDown(int integer)
{
 while (integer >= 1)
 {
 System.out.println(integer);
 integer--;
 } // end while
} // end countDown
```

两个方法有类似的样子。两个方法都将 `integer` 值与 1 进行比较，但递归版本使用的是一个 `if` 语句，而迭代版本使用的是一个 `while` 语句。两个方法都显示 `integer`。两个方法都计算 `integer - 1`。

**!** 程序设计技巧：迭代方法含有一个循环。递归方法调用自己。虽然有些递归方法也含有一个循环且调用自身，但如果我在递归方法中写 `while` 语句，一定要确保你不是想写一个 `if` 语句。

**?** STUDY 学习问题 1 写递归的 `void` 方法，跳过  $n$  行输出，这里  $n$  是一个正整数。使用 `System.out.println()` 跳过一行。

学习问题 2 使用伪代码描述一个递归算法，画指定数目的同心圆。最内圈的圆应该有给定的直径。其他每个圆的直径应该是其内侧紧邻它的圆直径的  $4/3$  倍。

## 跟踪递归方法

现在让我们来跟踪段 9.3 中给出的方法 `countDown`:

```
public static void countDown(int integer)
{
 System.out.println(integer);
 if (integer > 1)
 countDown(integer - 1);
} // end countDown
```

为简单起见，假定在定义 `countDown` 的类的 `main` 方法内，用下列语句调用这个方法：

```
countDown(3);
```

这个调用与其他的对非递归方法的调用是一样的。实参 3 要拷贝给形参 `integer`，且执行下列语句：

```
System.out.println(3);
if (3 > 1)
 countDown(3 - 1); // First recursive call
```

显示含有数字 3 的一行，然后递归调用 `countDown(2)`，如图 9-2a 所示。调用 `countDown(3)` 的执行暂停，直到得到 `countDown(2)` 的结果时为止。

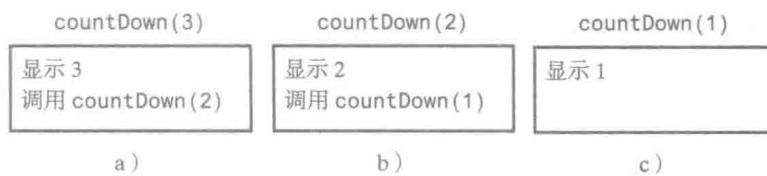


图 9-2 方法调用 `countDown(3)` 的效果

9.9 继续我们的跟踪, `countDown(2)` 导致执行下列语句:

```
System.out.println(2);
if (2 > 1)
 countDown(2 - 1); // Second recursive call
```

显示含有数字 2 的一行, 然后递归调用 `countDown(1)`, 如图 9-2b 所示。调用 `countDown(2)` 的执行暂停, 直到得到 `countDown(1)` 的结果时为止。

调用 `countDown(1)` 导致执行下列语句:

```
System.out.println(1);
if (1 > 1)
```

显示含有数字 1 的一行, 如图 9-2c 所示, 不再发生其他的递归调用。方法执行完成并返回给客户。

图 9-3 说明了使用参数 3 首次调用 `countDown` 时的事件序列。编号箭头表示递归调用及从方法返回的次序。显示 1 后, 方法执行完毕并返回到调用 `countDown(2 - 1)` 后的位置 (箭头 4 处)。继续从那里执行, 方法返回到调用 `countDown(3 - 1)` 后的位置 (箭头 5 处)。最后, 返回到 `main` 中最初递归调用后的位置 (箭头 6 处)。

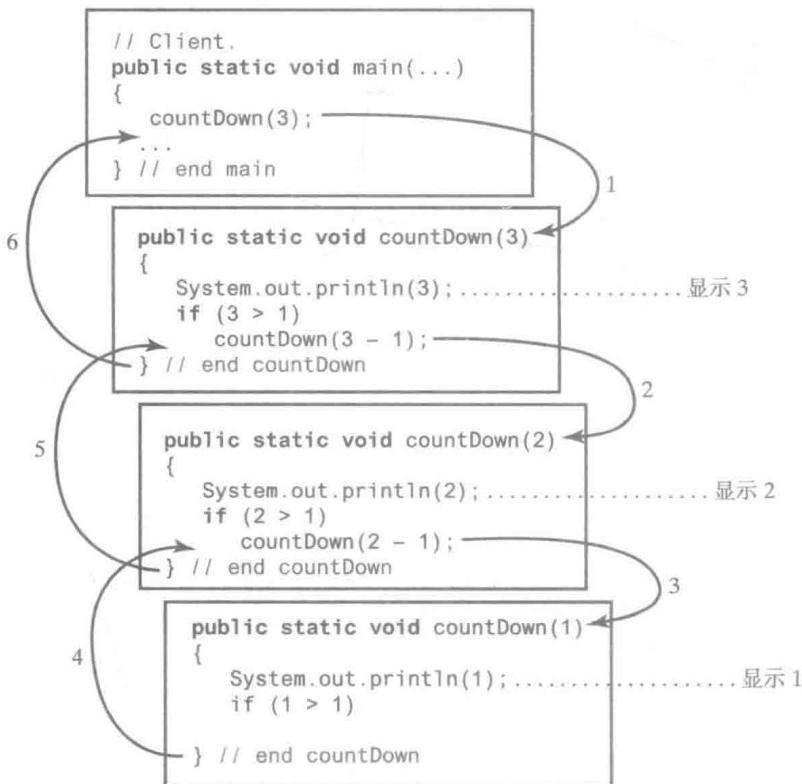


图 9-3 跟踪 `countDown(3)` 的执行过程

虽然跟踪这些方法时返回似乎只是个形式, 没为我们提供什么有用的信息, 但这是任何跟踪过程中的重要部分, 因为有些递归方法要做的远不仅是返回到它们的调用方法。马上就会看到一个这样的方法。

9.10 图 9-3 表示了方法 `countDown` 的多个拷贝。但实际上并不存在多个拷贝。对方法的每次调用——递归或非递归, Java 都要记录下方法执行的当前状态, 包括它的参数值和局部变

量的值，及当前指令的位置。如第 5 章段 5.22 所述，每个记录称为一个活动记录，它提供了运行期间方法状态的快照。记录放入程序栈中。栈按时间先后组织这些记录，所以当前正在执行的方法的记录在栈顶。这种机制下，Java 可以暂停递归方法的运行，并用新的变量值再次调用它。图 9-3 中的方框大致对应于活动记录，不过图中没有按它们在栈中出现的次序来显示。在 main 方法中调用 countDown(3) 时得到的活动记录栈如图 9-4 所示。

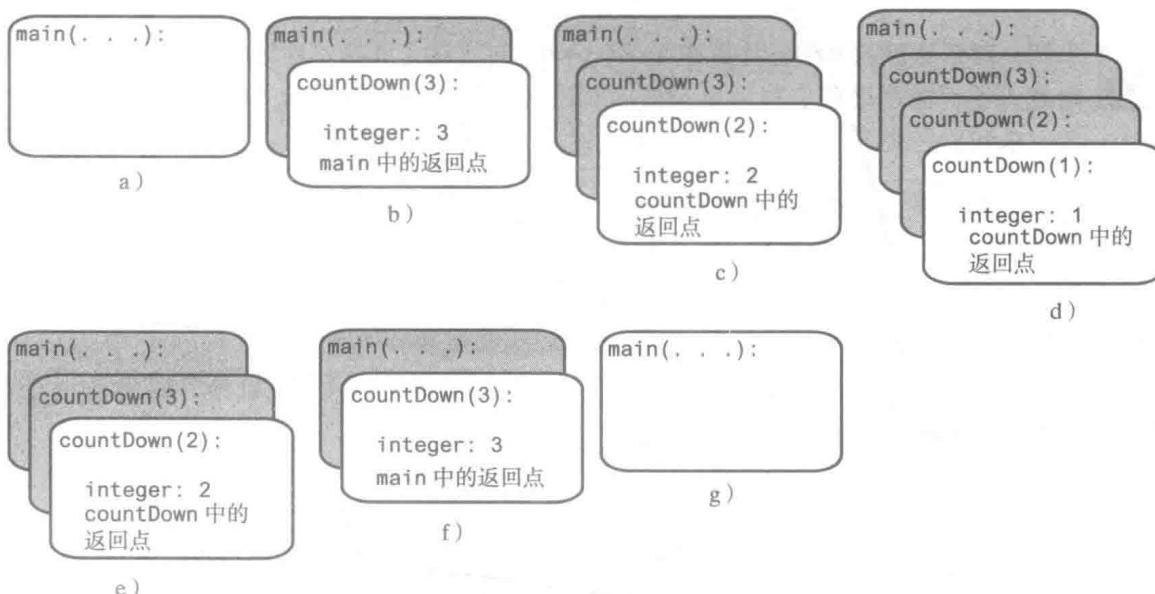


图 9-4 调用 `countDown(3)` 执行期间的活动记录栈

### 注：活动记录栈

每次调用一个方法时都生成一个活动记录，它获取方法的运行状态，并被放到程序栈中。第 5 章中的图 5-13 说明的是当 `methodA` 调用不同的方法 `methodB` 时的程序栈。不过，这些方法不必非得不一样。即，程序栈能让运行时环境执行递归方法。每次调用任何方法都产生一个活动记录并记入程序栈中。递归方法的活动记录并没什么特殊之处。

**注：**递归方法一般比迭代方法使用更多的内存，因为每次递归调用都要产生一个活动记录。

### 程序设计技巧：栈溢出

进行许多次递归调用的递归方法，会在程序栈中放很多个活动记录。递归调用太多可能会用掉程序栈中可用的所有内存，而使得栈满。结果，会报出错信息“栈溢出”。无穷递归或较大规模问题都可能导致这个错误。

**学习问题 3** 写一个递归的 void 方法 `countUp(n)`，从 1 加到 `n`，这里 `n` 是一个正整数。提示：在显示信息之前进行递归调用。

## 返回一个值的递归方法

9.11

前一节的递归方法 `countDown` 是一个 `void` 方法。值方法也可以是递归的。段 9.4 给出的成功递归的原则也适用于值方法，这一点可作为附加说明。回忆一下，递归方法必须含有如 `if` 这样的语句，它在几种情形中进行选择。有些情形导向递归调用，但至少有一种情形没有递归调用。对于一个值方法，这些情形中的每一种都必须提供一个值作为方法的返回值。

9.12

 **示例：对任意整数  $n > 0$ ，计算和  $1+2+\dots+n$ 。** 对于这个问题，所给的输入值是整数  $n$ 。由此入手，能帮助我们找到更小的问题，因为更小的问题的输入也还是一个单一整数。求和总是从 1 开始，所以可以认为这是更小的一个问题。

假定我给你一个正整数  $n$ ，要求你计算前  $n$  个整数的和。你必须要求一位朋友，对某个正整数  $m$  来计算前  $m$  个整数的和。 $m$  应该是多少呢？当然，如果你朋友计算了  $1+\dots+(n-1)$ ，你简单地将  $n$  加到这个和上就得到了你的结果。所以如果 `sumOf(n)` 是返回前  $n$  个整数和的方法调用，将  $n$  加到你朋友得到的和上的表达式就是 `sumOf(n-1) + n`。

哪个小问题可能是基础情形呢？即  $n$  是什么值时你能立即知道和？答案可能是 1。如果  $n$  是 1，则要得到的和是 1。

有了这些想法，可以写下面的方法：

```
/** @param n An integer > 0.
 * @return The sum 1 + 2 + ... + n. */
public static int sumOf(int n)
{
 int sum;
 if (n == 1)
 sum = 1; // Base case
 else
 sum = sumOf(n - 1) + n; // Recursive call
 return sum;
} // end sumOf
```

9.13

方法 `sumOf` 的定义符合成功递归的设计原则。所以，应该自信，方法能正确工作而不需要跟踪它的执行过程。不过，此处的跟踪是有益的，因为这不仅能让你明白值递归方法是如何工作的，还能说明递归调用完成后发生动作。

假定我们用下面的语句调用这个方法：

```
System.out.println(sumOf(3));
```

它会进行如下的计算：

- 1) `sumOf(3)` 是 `sumOf(2)+3`; `sumOf(3)` 暂停执行，开始执行 `sumOf(2)`。
- 2) `sumOf(2)` 是 `sumOf(1)+2`; `sumOf(2)` 暂停执行，开始执行 `sumOf(1)`。
- 3) `sumOf(1)` 返回 1。

一旦到达基础情形，从最近的方法开始恢复暂停的运行。所以 `sumOf(2)` 返回  $1+2$ （或者 3）；然后 `sumOf(3)` 返回  $3+3$ （或者 6）。图 9-5 说明了这个计算过程。



**学习问题 4** 写一个递归的值方法，计算整数  $1 \sim n$  的乘积，其中  $n > 0$ 。

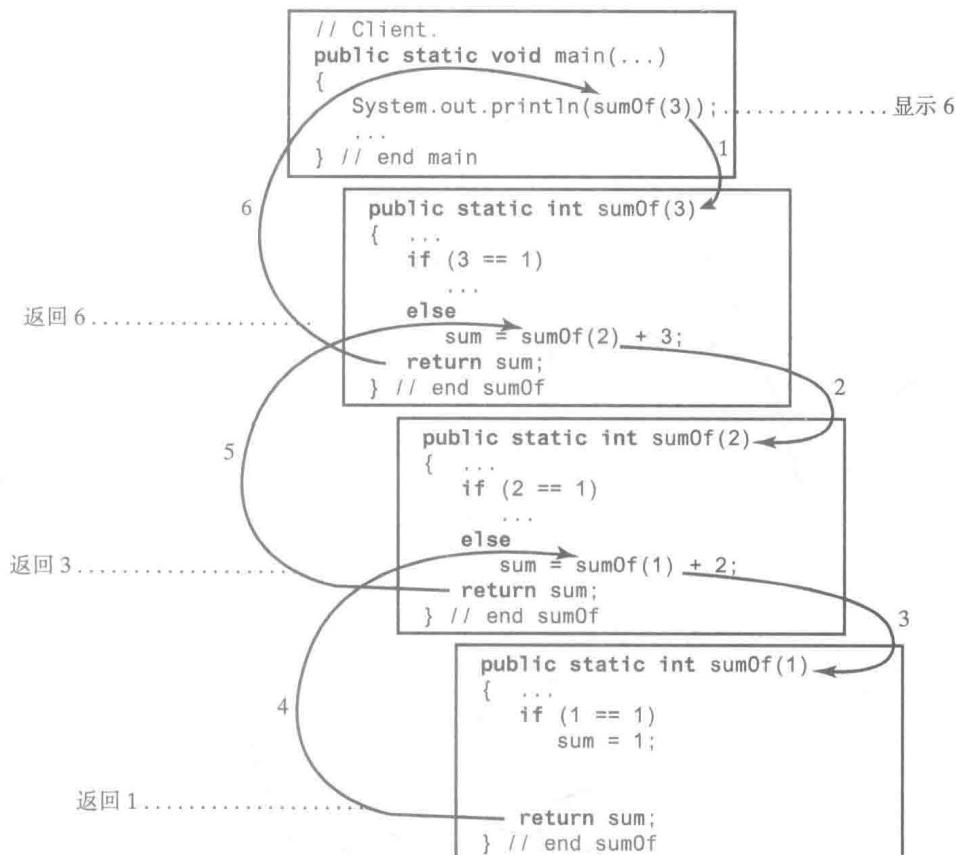


图 9-5 跟踪 sumOf(3) 的执行

### 注：应该跟踪递归方法吗？

我们已经展示给你如何跟踪递归方法的执行，向你说明了递归是如何工作的，且让你明白了一般的编译程序是如何实现递归的。你到底应不应该跟踪一个递归方法呢？通常情况下不需要。你肯定不应该跟踪一个正在写的递归方法。如果方法尚未完成，你的跟踪也完成不了，而且可能会让你困惑。如果递归方法不能正常工作，遵照下面这个程序设计技巧中所给的建议去做。跟踪递归方法应该仅作为最后的手段。

### 程序设计技巧：调试递归方法

如果递归方法不能正常工作，则回答下列问题。任何“不”的回答都应该指示给你错误之所在。

- 方法至少有一个输入值吗？
- 方法含有测试输入值的语句，且能导向不同情形吗？
- 考虑了所有可能的情形吗？
- 至少有一种情形导致了至少一次的递归调用吗？
- 这些递归调用涉及了更小的实参、更小的任务或者接近于解决方案的任务吗？
- 如果这些递归调用产生或返回了正确的结果，则方法产生或返回正确结果了吗？
- 是不是至少有一种情形即基础情形没有递归调用？
- 基础情形足够吗？

- 每个基础情形都能得到对应于这种情形的正确结果吗？
- 如果方法返回一个值，则每种情形都返回一个值了吗？

**9.14** 前一个例子是简单的，所以你可以学习递归方法的结构。因为可以使用简单的迭代方式求解这些问题，故实际中真的应该用递归方式吗？这些递归方法本身并无不妥。但是，就目前经典系统中执行递归方法的方式来说，对于大的  $n$  值很可能会导致栈溢出。对这些简单示例，采用迭代方法求解将不会有这样的麻烦，且更容易编写程序。但要知道，未来的计算机系统可能会毫不费力地运行这些递归方法。

## 递归处理数组

本书的后面，我们将讨论如何在数组中查找一个具体的项。还要看看排序（sort）算法，或称按升序或降序重排数组中的项。一些有效的查找和排序算法常常是递归的。本节，我们递归地处理数组，这些方法对后面的讨论是有帮助的。我们选择一个简单的任务——显示数组中的整数作为示例，这样你可以将注意力集中到递归上而不是分散到任务本身。本书的后面及本章结尾的练习中，我们将考虑更复杂的任务。

**9.15** 假定有一个整数数组，想写一个显示数组元素的方法。方法将显示数组下标在 `first` 到 `last` 范围内的各元素中的整数，这样我们就可以显示数组的全部或是一部分内容。故方法的说明如下所示。

```
/** Displays the integers in an array.
 * @param array An array of integers.
 * @param first The index of the first integer displayed.
 * @param last The index of the last integer displayed,
 * 0 <= first <= last < array.length. */
public static void displayArray(int[] array, int first, int last)
```

这个任务很简单，可以快速地使用迭代来实现。但是你可能无法想象，我们还可以用不同的递归方法来实现它。我们可以而且将会这样做。

**9.16** 从 `array[first]` 开始。迭代方法常常会从第一个元素 `array[first]` 开始，很自然，我们的第一个迭代方法也是从那个元素开始的。如果我要求你显示数组，那么你可以显示 `array[first]` 中的整数，然后要求一位朋友来显示数组中的其余元素。显示数组的其余元素是比显示整个数组更小的任务。如果你只需显示一个元素——即如果 `first` 和 `last` 相等，那么你不需要让朋友帮忙。这是基础情形。所以我们可以写出方法 `displayArray`，如下所示。

```
public static void displayArray(int array[], int first, int last)
{
 System.out.print(array[first] + " ");
 if (first < last)
 displayArray(array, first + 1, last);
} // end displayArray
```

为简单起见，假定整数能放在一行中。注意，客户在调用 `displayArray` 后可以使用 `System.out.println()` 来换行。

**9.17** 从 `array[last]` 开始。虽然看起来很奇怪，但是确实可以从数组的最后面的一个元素开始，且仍从头开始显示数组。不是立即显示数组的最后面的整数，而是要求朋友显示数组中的其他内容。在显示了从 `array[first]` 到 `array[last - 1]` 中的整数之后，再来显示 `array[last]` 中的整数。得到的输出应该与前一段是一样的。

采用这个思路，实现的方法代码如下所示。

```

public static void displayArray(int array[], int first, int last)
{
 if (first <= last)
 {
 displayArray(array, first, last - 1);
 System.out.print(array[last] + " ");
 } // end if
} // end displayArray

```

将数组分半。递归处理数组的常用方法是将数组分为两部分。然后分别处理每一部分。9.18  
因为每个部分都是一个数组，且小于原数组，故每个都定义了递归处理的更小的问题。前两个示例也是将数组划分为两部分，但其中一部分仅包含一个元素。现在，我们将数组划分为大致相等的两个部分。为了划分数组，要找到位于或接近于数组中间位置的元素。这个元素的下标是

```
int mid = (first + last) / 2;
```

图 9-6 显示了两个数组及它们的中间元素。假定将  $array[mid]$  放在数组的左“半”部分，如图 9-6 所示。在图 9-6a 中，数组的两段在长度上是相等的；而在图 9-6b 中，它们不等。长度上小小的差别没什么关系。

再次强调，基础情形是含一个元素的数组。无须帮助就能显示它。但如果数组含有多个元素，则你需要将它划分为两半。然后要求一位朋友显示一半，另一位朋友显示另外一半。当然，这两位朋友代表下面方法中的两次递归调用：

```

public static void displayArray(int array[], int first, int last)
{
 if (first == last)
 System.out.print(array[first] + " ");
 else
 {
 int mid = (first + last) / 2;
 displayArray(array, first, mid);
 displayArray(array, mid + 1, last);
 } // end if
} // end displayArray

```

**学习问题 5** 假定数组的中间元素不属于数组两部分中的任何一个。这样，你可以递归地显示左半部分，显示中间元素，然后再递归地显示右半部分。如果做这些修改，则 `displayArray` 将如何实现？

**注：**当递归处理数组时，可以将它划分为两部分。例如，第一个元素或最后一个元素可以是一部分，数组的其余元素是另一部分。或者将数组分为两半，或采用其他的方式。

**注：找到数组的中点**

为计算数组中间元素的下标，可以用语句

```
int mid = first + (last - first) / 2;
```

来替代

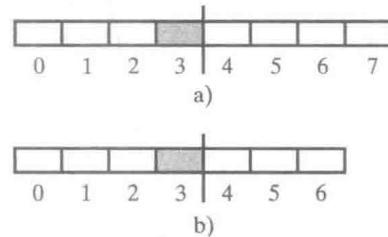


图 9-6 将中间元素含在左半部分的两个数组

```
int mid = (first + last) / 2;
```

如果查找至少含有 $2^{30}$ (大约10亿)个元素的数组，则first与last的和将超出最大可能的整数值 $2^{31}-1$ 。所以，计算first+last时将溢出为负数，mid得到的结果为一个负值。如果这个负的mid值用于数组下标，则会发生ArrayIndexOutOfBoundsException异常。而根据代数推导可知first + (last - first)/2等于(first + last)/2，可以避免这个错误。

9.19

**显示一个包。**在第2章中，使用数组实现了ADT包。假定类ArrayBag有一个方法display来显示包的内容。虽然可以迭代地定义这个方法，不过我们使用递归来定义。



### 注：display应该有参数吗？

了解到仅有ArrayBag的客户会调用display。客户不知道包的表示细节，所以不会传给display任何参数。如果myBag是ArrayBag的实例，则通过myBag.display()就可以显示它的内容。所以display没有参数。

因为方法display没有参数，所以它必须调用另一个方法——displayArray，它有参数且显示包项所在的数组。调用displayArray时的实参是，表示第一个下标的0和表示最后一个下标的numberOfEntries-1，其中，numberOfEntries是包类的数据域。因为包项所在的数组bag是实现包的类的数据域，所以它不必作为displayArray的参数。因为displayArray必须有关于包项数组的具体数据——通过其参数得到，故它必须是私有的。最后，因为display不是静态方法，所以displayArray也不是静态的。

我们可以使用前面给出的任何一个版本的displayArray。不过，我们要在每行显示一个对象，而不是在一行中显示多个整数。使用段9.16中介绍的技巧，修改方法如下。

```
public void display()
{
 displayArray(0, numberOfEntries - 1);
} // end display

private void displayArray(int first, int last)
{
 System.out.println(bag[first]);
 if (first < last)
 displayArray(first + 1, last);
} // end displayArray
```



**注：**作为实现ADT的组成部分的递归方法常常是私有的，因为要使用这个方法，需要了解底层数据结构。这样的方法不适合作为ADT的操作。

## 递归处理链表

9.20

现在用图来说明对结点链表执行简单任务的递归处理，比如显示链表中的数据。我们再次实现ADT包的方法display，但这次换作使用第3章中介绍的链式实现。这个实现定义了域firstNode，它指向链表中的首结点。

将链表分段不如划分数组那样简单，因为不从头遍历链表就不能访问任何结点。所以，第一个方法显示首结点中的数据，然后递归地显示链表中其余的数据。所以如段9.19中所做的那样，display将调用一个私有的递归方法。将方法命名为displayChain。作为一个

递归方法，`displayChain`需要一个输入值。这个输入值应该代表这个链表，所以将指向链表中首结点的引用作为传给`displayChain`的参数。

假定将`displayChain`的参数命名为`nodeOne`，则`nodeOne.getData()`是首结点中的数据，而`nodeOne.getNextNode()`是指向链表其余部分的引用。基础情形是什么呢？虽然单元素的数组是`displayArray`的很好的基础情形，但这里使用一个空链表作为基础情形更简单，因为我们只需要让`nodeOne`与`null`进行比较即可。由此，方法`display`和`displayChain`的实现如下所示。

```
public void display()
{
 displayChain(firstNode);
} // end display

private void displayChain(Node nodeOne)
{
 if (nodeOne != null)
 {
 System.out.println(nodeOne.getData()); // Display first node
 displayChain(nodeOne.getNextNode()); // Display rest of chain
 } // end if
} // end displayChain
```

 注：当写一个方法递归处理结点链表时，可以做以下工作。

- 使用指向链表中首结点的引用作为方法的参数。
- 处理首结点，然后处理链表中的其余结点。
- 当参数值是`null`时停止。

**反向显示链表。**假定你想以反序遍历结点链表。具体来说，假定你想显示最后一个结点中的对象，然后是倒数第二个结点中的对象，等等，即向链头方向前进。因为每个结点指向下一个结点但没有指向前一个结点，故使用迭代来完成这个任务是困难的。可以遍历到最后一个结点，显示它的内容，然后回到开头再遍历到倒数第二个结点，等等。但是，很明显，这是一个乏味且耗时的方法。换一种做法，可以仅遍历一次链表并保存指向每个结点的引用，然后用这些引用以反序来显示链表的结点中的对象。递归方法就可以这样做。

如果一位朋友可以以反序显示从第二个结点开始的子链表中的结点，那么你可以显示首结点，从而能完成任务。下列递归方案实现了这个思想。

```
public void displayBackward()
{
 displayChainBackward(firstNode);
} // end displayBackward

private void displayChainBackward(Node nodeOne)
{
 if (nodeOne != null)
 {
 displayChainBackward(nodeOne.getNextNode());
 System.out.println(nodeOne.getData());
 } // end if
} // end displayChainBackward
```

 注：采用递归方式以反序遍历结点链表，比起采用迭代实现要简单些。



### 学习问题 6 对含有 3 个结点的链表，跟踪前一个方法 displayBackward 的执行过程。

## 递归方法的时间效率

第 4 章展示了如何使用大  $O$  符号估算算法的时间需求。在选择合适的增长率函数时，第一步是对算法的主要操作进行计数。对于要评测的迭代示例，处理过程是简单的。这里我们使用更形式化的技术来估算迭代算法的时间需求，并由此来选择正确的增长率函数。

### countDown 的时间效率

**9.22** 考虑段 9.3 给出的 countDown 方法作为第一个示例。从一个给定的整数倒数到 1，这个问题的长度与所给的整数直接相关。因为第 4 章使用  $n$  来表示问题长度，所以我们将 countDown 中的参数 integer 重命名为  $n$ ，以简化我们的讨论。下面是修改后的方法。

```
public static void countDown(int n)
{
 System.out.println(n);
 if (n > 1)
 countDown(n - 1);
} // end countDown
```

当  $n$  为 1 时，countDown 显示 1。这是基础情形，需要常数级的时间。当  $n > 1$  时，方法执行 `println` 语句及进行比较时都需要常数级的时间。另外，它需要时间去解决由递归调用所表示的更小的问题。如果令  $t(n)$  表示 `countDown(n)` 的时间需求，则以上讨论的结果可写为

$$\begin{aligned} t(1) &= 1 \\ t(n) &= 1 + t(n - 1) \quad \text{对于 } n > 1 \end{aligned}$$

表示  $t(n)$  的方程称为递推关系（recurrence relation），因为函数  $t$  的定义中又含有自身，即递推。我们需要一个不由自己定义自己的表达式来表示  $t(n)$ 。找到这种表达式的一种办法是，挑选一个  $n$  值，写出  $t(n)$ 、 $t(n-1)$  等的方程，直到到达  $t(1)$ 。从这些方程中，我们应该能猜出表示  $t(n)$  的合适的表达式。然后所需的就是证明我们是正确的。实际工作比听上去更简单些。

**9.23** 求解一个递推关系。为求解前面关于  $t(n)$  的递推关系，从  $n=4$  开始。得到下面的方程序列：

$$\begin{aligned} t(4) &= 1 + t(3) \\ t(3) &= 1 + t(2) \\ t(2) &= 1 + t(1) = 1 + 1 = 2 \end{aligned}$$

在  $t(3)$  的方程中，用 2 替代  $t(2)$ ，得到

$$t(3) = 1 + 2 = 3$$

在  $t(4)$  的方程中，用 3 替代  $t(3)$ ，得到

$$t(4) = 1 + 3 = 4$$

似乎得到

$$t(n) = n \quad \text{对于 } n \geq 1$$

我们可以从一个较大的  $n$  值开始，得到同样的结果，这让我们相信这是对的。但我们需要证明这个结果对于任意的  $n \geq 1$  都是对的。这个不难做。

证明  $t(n) = n$ 。为证明对任意的  $n \geq 1$  有  $t(n)=n$ ，我们从  $t(n)$  的递推关系开始，因为我们知道下列关系是成立的：

$$t(n) = 1 + t(n-1) \quad \text{对于 } n > 1$$

我们需要替换掉方程右侧的  $t(n-1)$ 。如果当  $n>1$  时有  $t(n-1)=n-1$ ，则当  $n>1$  时下列关系是正确的：

$$t(n) = 1 + n - 1 = n$$

所以，如果我们能找到整数  $k$ ，满足方程  $t(k)=k$ ，则下一个整数也将满足它。用类似的推理过程，对于大于  $k$  的所有整数，方程都是正确的。因为已给条件  $t(1)=1$ ，所以所有大于 1 的整数都满足方程。这个证明是归纳法证明（proof by induction）的例子。

最后，我们知道 `countDown` 的时间需求由函数  $t(n)=n$  给出。所以方法是  $O(n)$  的。



**学习问题 7** 段 9.12 给出的 `sumOf` 方法的大  $O$  表示是多少？

**学习问题 8** 对某个实数  $x$  及整数幂次  $n \geq 0$ ，计算  $x^n$  时有一个简单的递归解法：

$$x^n = x \cdot x^{n-1}$$

$$x^0 = 1$$

- 描述这个算法的时间需求的递推关系是什么？
- 求解这个递推关系，找到这个算法的大  $O$  表示。

## 计算 $x^n$ 的时间效率

我们可以使用比学习问题 8 中提到的方法更有效率的方法，对于某个实数  $x$  和整数幂次  $n \geq 0$ ，计算  $x^n$ 。为减少递归调用的次数，从而也减少乘法的次数，可以将  $x^n$  表示为：

$$x^n = (x^{n/2})^2 \quad \text{当 } n \text{ 是正偶数时}$$

$$x^n = x(x^{(n-1)/2})^2 \quad \text{当 } n \text{ 是正奇数时}$$

$$x^0 = 1$$

这个计算可以由方法 `power(x, n)` 实现，它含有递归调用 `power(x, n/2)`。因为在 Java 中整除是截断结果，所以不管  $n$  是偶数还是奇数，这个调用都是合适的。所以 `power(x, n)` 将调用 `power(x, n/2)` 一次，然后将结果进行平方，如果  $n$  是奇数再将平方乘上  $x$ 。这些乘积都是  $O(1)$  操作。所以 `power(x, n)` 的运行时间与递归调用的次数成正比。

表示递归调用次数的递推关系，即计算  $x^n$  的方法的时间需求是

$$t(n) = 1 + t(n/2) \quad \text{当 } n \geq 2 \text{ 时}$$

$$t(1) = 1$$

$$t(0) = 1$$

再次说明， $n/2$  将截断为一个整数。

因为递推关系中涉及  $n/2$ ，所以我们选择 2 的幂次——例如 16——作为  $n$  的初始值。则有下列方程序列：

$$t(16) = 1 + t(8)$$

$$t(8) = 1 + t(4)$$

$$t(4) = 1 + t(2)$$

$$t(2) = 1 + t(1)$$

通过反复替代，得到：

$$t(16) = 1 + t(8) = 1 + (1 + t(4)) = 2 + (1 + t(2)) = 3 + (1 + t(1)) = 4 + t(1)$$

因为  $16=2^4$ , 故  $4 = \log_2 16$ 。这个条件再加上基础情形  $t(1) = 1$ , 我们可猜想

$$t(n) = 1 + \log_2 n$$

9.27 现在需要证明, 对于  $n \geq 1$ , 这个猜想确实是对的。对于  $n = 1$ , 它是正确的, 因为

$$t(1) = 1 + \log_2 1 = 1$$

对于  $n > 1$ , 我们知道  $t(n)$  的递推关系

$$t(n) = 1 + t(n/2)$$

是正确的。记住,  $n/2$  将截断为一个整数。

现在需要替换掉  $t(n/2)$ 。如果我们猜测对于所有的  $n < k$ , 都有  $t(n) = 1 + \log_2 n$ , 则我们将有  $t(k/2) = 1 + \log_2 (k/2)$ , 因为  $k/2 < k$ 。所以

$$\begin{aligned} t(k) &= 1 + t(k/2) \\ &= 1 + (1 + \log_2 (k/2)) \\ &= 2 + \log_2 (k/2) \\ &= \log_2 4 + \log_2 (k/2) \\ &= \log_2 (4k/2) \\ &= \log_2 (2k) \\ &= \log_2 2 + \log_2 k \\ &= 1 + \log_2 k \end{aligned}$$

总之, 我们假定, 对所有的  $n < k$ , 有  $t(n) = 1 + \log_2 n$ , 并展示了  $t(k)=1+\log_2 k$ 。所以对所有的  $n \geq 1$ , 有  $t(n)=1+\log_2 n$ 。因为 power 的时间需求由  $t(n)$  表示, 所以方法是  $O(\log n)$  的。

## 尾递归

9.28 当递归方法执行的最后一个动作是递归调用时发生尾递归 (tail recursion)。例如, 下面这个来自段 9.6 中的方法 countDown 是尾递归:

```
public static void countDown(int integer)
{
 if (integer >= 1)
 {
 System.out.println(integer);
 countDown(integer - 1);
 } // end if
} // end countDown
```

方法中的尾递归只是用改变的参数和变量重复了方法的逻辑。所以你可以使用迭代执行相同的重复。将尾递归方法转为迭代方法, 通常是一个简单的过程。以刚给出的方法 countDown 为例, 来看看如何转换递归方法。首先, 我们将 if 语句用 while 语句来替换。然后, 不是进行递归调用, 而是将实参 `integer - 1` 赋给方法的形参 `integer`。完成这些, 即得到方法的迭代版本, 如下所示。

```
public static void countDown(int integer)
{
 while (integer >= 1)
 {
 System.out.println(integer);
 integer = integer - 1;
 } // end while
} // end countDown
```

这个方法本质上与段 9.7 给出的迭代方法是一样的。

因为将尾递归转换为迭代通常不复杂，所以，除了 Java 之外的有些语言自动将尾递归方法转为迭代方法，为的是节省递归带来的开销。这些开销主要是涉及内存，而不是时间。如果必须要节省空间，就应该考虑用迭代来替代递归。

 注：在尾递归方法中，最后一个动作是递归调用。这个调用执行的重复部分可以使用迭代来完成。将尾递归方法转为迭代方法，常常是一个简单的过程。

## 使用栈来替代递归

使用迭代来替代递归的一个方法是模拟程序栈。事实上，我们可以使用一个栈来替代递归，从而实现递归算法。我们以段 9.18 中所给的方法 `displayArray` 为例，介绍将递归方法转换为迭代方法的过程。为了能说明问题，将 `displayArray` 修改为类内的一个非静态方法，并带有一个数组作为数据域。做了这些修改后，方法如下所示。

```
public void displayArray(int first, int last)
{
 if (first == last)
 System.out.println(array[first] + " ");
 else
 {
 int mid = first + (last - first) / 2; // Improved calculation of midpoint
 displayArray(first, mid);
 displayArray(mid + 1, last);
 } // end if
} // end displayArray
```

通过使用一个栈来模拟程序栈，可以将前一段给出的递归方法 `displayArray` 替换为迭代版本。为此，我们在方法内创建一个栈，作为局部变量使用。将类似于段 9.10 中描述的活动记录的对象入栈。Java 程序栈中的活动记录含有方法的实参、它的局部变量和指向当前指令的引用。因为对方法 `displayArray` 的两个递归调用是连续的，故不需要在活动记录中保存程序计数器的值来区分它们。但是这个简化对一般的情况不成立。

为表示一条记录，我们需要定义一个类，就本例来讲，要有对应于方法实参的数据域 `first` 和 `last`。如果我们让类定义在 `displayArray` 所在的类内，则下列简单的类就足够了。

```
private class Record
{
 private int first, last;
 private Record(int firstIndex, int lastIndex)
 {
 first = firstIndex;
 last = lastIndex;
 } // end constructor
} // end Record
```

一般地，当方法开始运行时，它将一个活动记录压入程序栈中。当它返回时，从这个栈中弹出一个记录。我们让迭代的 `displayArray` 来维护自己的栈。当方法开始运行时，它应该将一个记录压入这个栈中。每次递归调用都应该这样做。当栈不空时，方法应该从栈中删除一个记录，并根据记录的内容来执行。当栈空时方法结束运行。

下面是使用我们刚刚描述的栈来实现的 `displayArray` 的迭代版本。

```
private void displayArray(int first, int last)
{
```

```

boolean done = false;
StackInterface<Record> programStack = new LinkedStack<>();
programStack.push(new Record(first, last));
while (!done && !programStack.isEmpty())
{
 Record topRecord = programStack.pop();
 first = topRecord.first;
 last = topRecord.last;
 if (first == last)
 System.out.println(array[first] + " ");
 else
 {
 int mid = first + (last - first) / 2;
 // Note the order of the records pushed onto the stack
 programStack.push(new Record(mid + 1, last));
 programStack.push(new Record(first, mid));
 } // end if
} // end while
} // end displayArray

```

这个方法并不总能得到简洁的程序。我们肯定能写一个比这个版本更容易理解的 `displayArray` 的迭代版本，并且不需要栈。但有时，一个简单的迭代版本并不是那么容易得到的；这种情况下，栈的方法就提供了一种可能的解决方案。你将会在第 25 章段 25.13 看到一个基于栈迭代的更有用的示例。

## 本章小结

- 递归是将问题划分为更小的同样问题的问题求解过程。
- 递归方法的定义必须含有能处理方法的输入（常常是一个形参）的逻辑，并导向不同的情形。其中的一个或多个情形是基础情形，或是终止情形，因为它们提供的是不再需要递归的答案。一个或多个情形中包括了方法的递归调用，通过求解“更小”版本的任务，而向基础情形迈进。
- 对方法的每次调用，Java 将方法形参和局部变量的值记录在活动记录中。记录被放到栈中，栈按时间顺序组织记录。最近入栈的记录是当前正在运行的方法的。这种方式下，Java 可以暂停递归方法的执行，并用新的实参值重新执行它。
- 递归方法处理一个数组时，常常将数组分成几部分。对方法的递归调用将处理数组的每个部分。
- 处理结点链表的递归方法，需要一个指向链表首结点的引用作为形参。
- 作为实现 ADT 的组成部分的递归方法常常是私有的，因为要使用这个方法，需要对底层数据结构的了解。尽管这样的方法不适合作为 ADT 的操作，但它可以被实现某个操作的公有方法来调用。
- 递推关系用函数自己来表示函数。可以使用递推关系来描述递归方法所做的事情。
- 当递归方法的最后一个动作是递归调用时出现尾递归。这个递归调用执行的重复部分可用迭代来完成。将尾递归方法转换为迭代方法，通常是一个简单的过程。
- 你可以使用栈替代递归来实现递归算法。这个栈模拟了程序栈的行为。

## 程序设计技巧

- 要写一个正确执行的递归方法，一般应该遵守下列设计原则：
  - ◆ 必须给方法一个输入值，通常作为实参给出。

- ◆ 方法定义中必须含有使用了该输入值并能导向不同情形的逻辑。一般这样的逻辑包含一个 `if` 语句或一个 `switch` 语句。
- ◆ 这些情形中的一个或多个，应该提供了不再需要递归的解决方案。这些是基础情形，或称终止情形。
- ◆ 一个或多个情形中必须包含对方法的递归调用。这些递归调用中应该含有一些步骤，通过使用“更小”的参数，或者说由方法完成的“更小”版本的任务的求解，在某种意义上逐步导向基础情形。
- 迭代方法包含一个循环。递归方法调用自己。虽然有些递归方法内含有循环并且调用自身，但如果你在递归方法内写一个 `while` 语句，要确定你不是要写一个 `if` 语句。
- 不检查基础情形或丢掉了基础情形的递归方法，不会正常终止。这种情况称为无穷递归。
- 递归调用太多会导致错误信息“stack overflow”（栈溢出）。这意味着活动记录的栈已经满了。本质上是方法使用了太多的内存。无穷递归或是大规模的问题容易引起这个错误。
- 不要使用在递归调用中重复求解同一问题的递归方案。
- 如果递归方法没有得到想要的结果，则回答下列问题。任何否定的答案都可能帮助你找到错误。
  - ◆ 方法至少有一个形参或输入值吗？
  - ◆ 方法含有测试形参或输入值的语句，且能导向不同情形吗？
  - ◆ 考虑了所有可能的情形吗？
  - ◆ 至少有一种情形导致至少一次的递归调用吗？
  - ◆ 这些递归调用涉及了更小的实参、更小的任务或者接近于解决方案的任务吗？
  - ◆ 如果这些递归调用产生或返回了正确的结果，那么方法产生或返回正确结果了吗？
  - ◆ 是不是至少有一种情形即基础情形没有递归调用？
  - ◆ 基础情形足够吗？
  - ◆ 每个基础情形都能得到对应于这种情形的正确结果吗？
  - ◆ 如果方法返回一个值，则每种情形都返回了一个值吗？

## 练习

1. 考虑方法 `displayRowOfCharacters`，它在一行内按指定的个数显示给定的任意字符。例如，调用

```
displayRowOfCharacters('*', 5);
```

将得到一行

\*\*\*\*\*

使用递归用 Java 语言实现这个方法。

2. 描述画同心圆的递归算法，给定最外层圆的直径。每个内层圆的直径是包含它的圆的直径的 3/4。最内层的圆的直径应该大于 1 英寸（1 英寸等于 2.54 厘米。——译者注）。
3. 写一个方法，要求用户输入 1 ~ 10（含）之间的一个整数。如果输入的数超出范围，方法将递归地

要求用户输入一个新值。

4. 正整数  $n$  的阶乘——表示为  $n!$ ——是  $n$  及  $n-1$  的阶乘的乘积。0 的阶乘是 1。写两个不同的递归方法，均返回  $n$  的阶乘。
5. 写一个递归方法，反向显示给定的数组。先考虑数组的最后一个元素。
6. 重做练习 5，但先考虑数组的第一个元素。
7. 重做练习 5 和练习 6，处理字符串而不是数组。
8. 一个回文是向前读和向后读都一样的字符串。例如，deed 和 level 都是回文。用伪代码写出算法，测试一个字符串是否是回文。使用 Java 语言将算法实现为一个静态方法。第 5 章的练习 11 和项目 3 要求你描述如何用栈来实现。
9. 写一个递归方法，统计结点链表中结点的个数。
10. 如果  $n$  是 Java 中的正整数，则  $n \% 10$  是其最右侧的一位数字，而  $n/10$  是  $n$  丢掉最右一位数字后得到的整数。使用这些已知，写一个递归方法，显示十进制的整数  $n$ 。现在，用新的基数替换 10，就可以显示 2 ~ 9 之间任何基数的值  $n$ 。修改你的方法适用于给定的基数。
11. 写 4 个不同的递归方法，每一个都计算整数数组中各整数的和。模仿段 9.15 到段 9.18 所给的 `displayArray` 方法及学习问题 5 中的描述。
12. 写一个递归方法，返回整数数组中的最小值。如果你将数组分为两部分——例如对半分——找到两部分中各自的最小值，整个数组中的最小值则是这两个整数中的较小者。因为你查找的是数组的一部分——例如从 `array[first]` 到 `array[last]`——故方法有数组及两个下标 `first` 和 `last` 这样 3 个形参将是方便的。对于这个问题，你可以参考段 9.18 中的 `displayArray`。
13. 对于下列方法，跟踪调用 `f(16)`，显示活动记录栈：

```
public int f(int n)
{
 int result = 0;
 if (f <= 4)
 result = 1;
 else
 result = f(n / 2) + f(n / 4);
 return result;
} // end f
```

14. 用伪代码写一个递归算法，找到 `Comparable` 对象数组中第二小的对象。使用 Java 语言将你的算法实现为一个静态方法。
15. 考虑第 2 章段 2.31 给出的 `ArrayBag` 类中私有的迭代方法 `getIndexOf`。修改这个方法，让其使用递归方法替代迭代方法。迭代定义中，按顺序每次检测数组中的一个项，直到找到需要的项，或是到达数组尾但没找到需要的项。假定方法已经检查了下标  $i$  处的项，但刚才所说的两种情况都没出现。递归步骤是从下标  $i+1$  处开始去查找数组的其他元素——一个更小的问题。方法需要  $i$  作为形参，因为方法是私有的，所以我们可以毫无顾虑地修改它的声明。但做这个修改需要找到并修改 `ArrayBag` 中其他方法中对 `getIndexOf` 的调用。
16. 考虑第 2 章段 2.20 中给出的 `ArrayBag` 类中的公有迭代方法 `getFrequencyOf`。写一个私有递归方法，让 `getFrequencyOf` 可以调用，并据此修改 `getFrequencyOf` 的定义。提示：虽然方法 `getIndexOf` 可能没有查看数组中的每一项，但 `getFrequencyOf` 必须要查看。递归方法有一个基础情形，且必须在两个递归步骤中选其一。
17. 针对第 3 章段 3.16 中给出的 `LinkedBag` 类中的 `getFrequencyOf` 方法，重做练习 16。
18. 考虑第 2 章给出的类 `ArrayBag`。为 `ArrayBag` 类实现方法 `equals`，让它调用一个私有的递归方法。
19. 如果

$$t(1) = 2$$

$$t(n) = 1 + t(n - 1) \quad \text{对于 } n > 1$$

找出不由自己表示的  $t(n)$  的表达式。使用归纳法证明你的结论是正确的。

20. 如果

$$t(1) = 1$$

$$t(n) = 2 \times t(n - 1) \quad \text{对于 } n > 1$$

找出不由自己表示的  $t(n)$  的表达式。使用归纳法证明你的结论是正确的。

21. 考虑一个棋盘，每个方块内印有美元金额。可以将棋子放在棋盘第一行的任何地方，然后按标准的对角线方式将棋子向最后一行的方向每次移动一行。一旦到达最后一行就结束。你将收集到与棋子首次移入的方块内所写的数值之和等量的钱数。

a. 给出递归算法，计算你能收集的最大值。

b. 给出一个迭代算法，使用栈来计算你能收集的最大值。

22. 考虑段 9.21 给出的反向显示结点链表中的内容的递归方法。也考虑练习 5 所描述的反向显示数组内容的递归方法。

a. 这两个方法的时间复杂度分别是多少？比较的结果如何？

b. 写一个反向显示结点链表中的内容的迭代方法。这个方法的时间复杂度是多少？与 a 中计算的复杂度相比较的结果如何？

23. 写一个静态递归方法，显示通过参数传给方法的字符串中各字符的全排列。例如字符序列 abc 有下列排列：acb、bac、bca、cab、cba。

## 项目

1. 下列算法找到正数的平方根：

```
Algorithm squareRoot(number, lowGuess, highGuess, tolerance)
newGuess = (lowGuess + highGuess) / 2
if ((highGuess - newGuess) / newGuess < tolerance)
 return newGuess
else if (newGuess * newGuess > number)
 return squareRoot(number, lowGuess, newGuess, tolerance)
else if (newGuess * newGuess < number)
 return squareRoot(number, newGuess, highGuess, tolerance)
else
 return newGuess
```

开始计算时，需要一个小于该数平方根的值 **lowGuess**，以及大于该数平方根的值 **highGuess**。可以使用 0 当作 **lowGuess**，用数本身当作 **highGuess**。参数 **tolerance** 控制结果的精度，它不依赖于数的大小。例如，当 **tolerance** 等于 0.000 05 时，计算 250 的平方根得到 15.81。这个结果有 4 位精度。

实现这个算法。

2. 考虑将栈  $S_1$  中的项进行排序的下列算法。首先创建两个空栈  $S_2$  和  $S_3$ 。任何时候，栈  $S_2$  都按大小有序保存项，最小值在栈顶。将  $S_1$  的栈顶项移到  $S_2$  中。 $S_1$  中的栈顶项  $t$  出栈，根据  $t$  的值，栈  $S_2$  出栈，并将出栈的值入栈  $S_3$  中，直到到达放置  $t$  的正确位置。然后将  $t$  入栈  $S_2$  中。接下来将  $S_3$  中的所有项再入栈  $S_2$  中。

写一个递归程序，实现这个算法。

3. 实现练习 21 描述的算法。

4. 假定你可以从含  $n$  项的数组中选择项。将选择的项放到大小为  $k$  的背包中。每个项有一个尺寸和一个价值。当然，你不能拿超出背包容量的项。你的目标是拿的项的价值最大。

a. 设计一个递归算法 **maxKnapsack** 来解决这个背包问题。算法的形参是背包、项的数组及要考虑的下一项在数组中的位置。这个算法选择放入背包的项，并返回含有所选项的背包。一个背包能

报告它的大小、它的内容、内容的值及内容的大小。

提示：如果数组中的任何项都还未被考虑，则获取数组中的下一项。可以忽略项，或是，如果它符合条件，将它放到背包中。递归调用对这两种情形中的每一种作出判断。比较由这些调用返回的背包，来看看哪种背包的内容有最大的价值。然后返回那个背包。

b. 写类 Knapsack 和 KnapsackItem。然后写一个程序，定义方法 maxKnapsack。程序应该读入背包的大小，然后读入每个可用项的大小、值及名字。下面是大小为 10 的背包的输入数据：

| 大小 | 值     | 名字               |
|----|-------|------------------|
| 1  | 50000 | rare coin        |
| 2  | 7000  | small gold coin  |
| 4  | 10000 | packet of stamps |
| 4  | 11000 | pearl necklace   |
| 5  | 12000 | silver bar       |
| 10 | 60000 | painting         |

显示这些项后，调用 maxKnapsack。然后显示所选项、它们的价值及总的价值。

5. 假定你正在给房间排时间表。给你一组活动，每个活动有开始时间和结束时间。如果两个活动不重叠，则它们是兼容的。例如，对于下列活动，活动 A 与活动 B 和活动 D 是兼容的，但与活动 C 不兼容：

| 活动 | 开始时间 | 终止时间 |
|----|------|------|
| A  | 1    | 2    |
| B  | 2    | 5    |
| C  | 1    | 3    |
| D  | 5    | 6    |

你的目标是安排兼容的活动，使得房间的利用率最高。

- a. 设计一个递归算法来解决这个房间调度问题。方法的签名是：

```
maxRoomUse(int startTime, int stopTime, Activity[] activities)
```

它返回二元对，分别是最大的使用小时数，及已安排的活动的数组。注意，startTime 是能被安排的活动的第一个时间，stopTime 是最后的时间，而 activities 是可能的活动的数组。

- b. 写类 Activity 和类 Schedule，代表 maxRoomUse 返回的二元对。然后写一个程序，定义方法 maxRoomUse。程序应该读入房间的开始时间和结束时间，后面是每个活动的开始和结束时间（一个活动占一行）。显示所给活动后，显示房间的最大的使用小时数，及安排的活动列表。

6. Java 的类 Graphics 中有下列方法，它在给定的两点间画线：

```
/** Draws a line between the points (x1, y1) and (x2, y2). */
public void drawLine(int x1, int y1, int x2, int y2)
```

Graphics 使用的坐标系以窗口的左上角作为原点。

写一个递归方法，画一把 12 英寸的直尺。标上英寸、半英寸、1/4 英寸及 1/8 英寸。半英寸的标记要小于英寸的标记。1/4 英寸的标记比半英寸的标记要小，以此类推。你的图不一定要和实际尺寸一样大。提示：在尺子的中间画一个标记，然后画出这个标记左边的尺子和右边的尺子。

7. 定义一个静态递归方法，方法的实参是表示罗马数字的一个字符串，返回等值的阿拉伯整数。  
 8. (游戏) 使用递归方法求解第 5 章项目 11 中的洞穴逃生问题。  
 9. (游戏) 求解第 5 章项目 10 描述的迷宫问题，使用下列递归算法而不是使用基于栈的算法。这个算法涉及回溯技术——也就是说，当你走到死胡同同时原路折回。第 14 章将进一步讨论回溯。

第 1 步：先检查你是否到达出口。如果是，则已经完成（一个非常简单的迷宫）；如果不是，则继续第 2 步。

第 2 步：调用 goNorth 方法尝试移动到正北面的方格中（继续第 3 步）。

第3步：如果 goNorth 成功，则已经完成。如果不成功，则调用 goWest 方法尝试移动到正西面的方格中（继续第4步）。

第4步：如果 goWest 成功，则已经完成。如果不成功，则调用 goSouth 方法尝试移动到正南面的方格中（继续第5步）。

第5步：如果 goSouth 成功，则已经完成。如果不成功，则调用 goEast 方法尝试移动到正东面的方格中（继续第6步）。

第6步：如果 goEast 成功，则已经完成。如果不成功，则也已经完成，因为从入口到出口不存在路径。

方法 goNorth 将从当前方格的北边方格开始检查所有的路径，方法如下。如果正北面的方格是空的、位于迷宫内且之前尚未被访问过，则移到该方格中并将它标记为路径的一部分。（注意，你从南面来的。）检查你是否已经位于出口。如果是，则已经完成。否则除了向南方向之外，尝试从当前方格出去的所有路径，以找到从当前方格通向出口的路径（向南走意味着你又折回到之前刚刚来的方格中）。这个操作的步骤如下。调用 goNorth；如果没有成功，则调用 goWest，如果没有成功，则调用 goEast。如果 goEast 没有成功，则标记这个方格已经访问过了，移回到南面的方格中，并返回。

下面伪代码描述了 goNorth 算法：

```
goNorth(maze, creature)
{
 if (北面的方格是空的、位于迷宫内且尚未访问过)
 {
 移到北面的方格中
 将方格标记为路径的一部分
 if (在出口)
 success = true
 else
 {
 success = goNorth(maze, creature)
 if (!success)
 {
 success = goWest(maze, creature)
 if (!success)
 {
 success = goEast(maze, creature)
 if (!success)
 {
 方格标记为已访问
 回溯到南面的方格中
 }
 }
 }
 }
 }
 else
 success = false
 return success
}
```

方法 goWest 将从当前方格的西边方格开始检查所有的路径，方法如下。如果正西面的方格是空的、位于迷宫内且之前尚未被访问过，则移到该方格中并将它标记为路径的一部分。（注意，你从东面来的。）检查你是否已经位于出口。如果是，则已经完成。否则，除了向东方向之外，尝试从当前方格出去的所有路径，以找到从当前方格通向出口的路径（向东走意味着你又折回到之前刚刚来的方格中）。这个操作的步骤如下。调用 goNorth；如果没有成功，则调用 goWest；如果没有成功，则调用 goSouth；如果 goSouth 没有成功，则标记这个方格已经访问过了，移回到东面的方格中，并返回。goEast 和 goSouth 方法类似于 goWest 和 goNorth 方法。

## 第 10 章 |

Data Structures and Abstractions with Java, Fifth Edition

# 线 性 表

先修章节：导论、附录 B、序言、Java 插曲 2、第 1 章

### 目标

学习完本章后，应该能够

- 描述 ADT 线性表
- 在 Java 程序中使用 ADT 线性表

线性表提供了组织数据的一种方法。我们可以有待办事项清单、礼品单、地址列表、杂货店名录，甚至是简单的清单。这些清单为我们提供了安排生活的一种有用方法，如图 10-1 所示。每个清单都有第一项、最后一项，通常在它们中间还有一些项。即列表中的项有一个位置：第一、第二，等等。项的位置对你可能是重要的，也可能是不重要的。当在列表中添加一项时，或许总将它加在最后，或许将它插入在列表中已有的两项之间。

一个线性表（list）是一个集合，本章将它形式化为一个 ADT。

这个周末有这么多事要  
处理——我应该列个清单。  
待办的事项  
1. 读第 10 章  
2. 给家里打电话  
3. 给 Sue 买贺卡



图 10-1 待办事项清单

## ADT 线性表的规范说明

像待办事项清单、礼品单、地址列表、杂货店名录这样的日常线性表都含有项，它们是

字符串。对这样的线性表我们能做什么？

- 10.1
- 一般地，可以在表尾（at the end）添加（add）一个新项。
  - 实际上，可以在任何地方（anywhere）添加（add）一个新项：在开头、在结尾或在项的中间。
  - 可以勾掉一项——即删除（remove）它。
  - 可以删除所有（remove all）的项。
  - 可以替换（replace）一项。
  - 可以查看或获取（get）给定位置的项。
  - 可以获取所有（get all）的项。
  - 可以查找而获知线性表是否含有（contain）某个项。
  - 可以对线性表中的项数进行计数（count）。
  - 可以查看线性表是否为空（empty）。

当处理线性表时，由你来决定项应处的位置。你可能并没有刻意留意它的具体位置：它是第 10 个？第 14 个？但是，当在程序中使用线性表时，识别一个具体的项的方便方法是使用项在线性表中的位置。它可能是第一个，即在位置 1 处，或是第二个（位置 2 处），等等。这个便利能让你更精确地描述或规范说明线性表上的操作。

为规范说明 ADT 线性表，我们描述它的数据并规范说明数据上的操作。与通常的其项为字符串的列表不一样，ADT 线性表更具一般性，并且它的项是有相同类型的对象。下面是 ADT 线性表的最初的规范说明：

- `add(newEntry)`: 在线性表尾添加一个新项。
- `add(newPosition, newEntry)`: 在线性表的给定位置添加一个新项。
- `remove(givenPosition)`: 从线性表中删除给定位置的项。
- `clear()`: 从线性表中删除所有的项。
- `replace(givenPosition, newEntry)`: 用给定的项替换线性表中给定位置的项。
- `getEntry(givenPosition)`: 获取线性表给定位置的项。
- `toArray()`: 按表中的当前次序获取线性表中的所有项。
- `contains(anEntry)`: 查看线性表中是否含有给定的项。
- `getLength()`: 得到线性表中的项数。
- `isEmpty()`: 查看线性表是否为空。

我们仅仅是开始规范说明线性表的这些操作，如前所述，而细节留给想象。有些例子能有助于我们更好地理解这些操作，以便我们能改进这些规范说明。在实现这些操作之前，我们需要让这些规范说明更加精确。

 **示例。**当首次声明一个新线性表时，它是空的，且长度为 0。如果 3 个对象——`a`、`b` 和 `c`——一次一个并按给定的次序添加到线性表尾，则线性表是下面这个样子的：

`a`  
`b`  
`c`

对象 `a` 是第一个，其位置为 1，`b` 在位置 2，而 `c` 在最后位置 3。<sup>Θ</sup> 为节省空间，我们有时将表的内容写在一行。例如，可能如下这样

`a b c`

来表示这个线性表。

下列伪代码表示对线性表 `myList` 进行了前面所述的这 3 次添加操作：

```
myList.add(a)
myList.add(b)
myList.add(c)
```

此时，`myList` 不再为空，所以 `myList.isEmpty()` 为假。因为线性表中含有 3 个项，故 `myList.getLength()` 是 3。注意到，在线性表尾添加项不会改变线性表中已有项的位置。图 10-2 说明了这些 `add` 操作，及下面将描述的其他操作。

10.2

<sup>Θ</sup> 有些人从 0 而不是从 1 开始为线性表中的项进行编号。

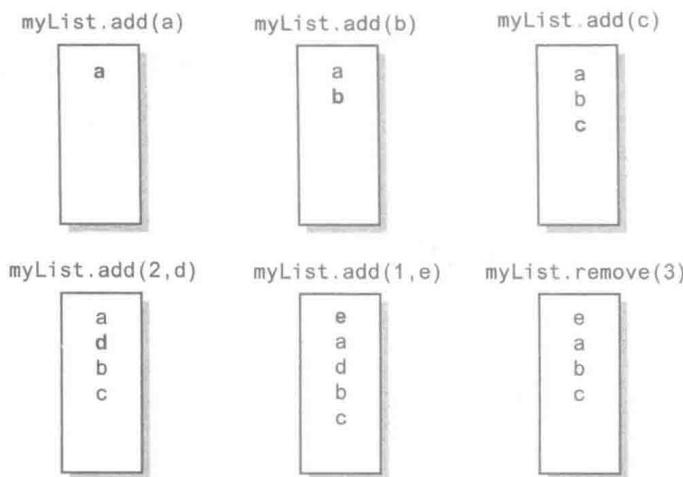


图 10-2 在初始为空的线性表中，进行 ADT 线性表操作的效果

10.4 现在假定，我们在含 a b c 的线性表的不同位置添加项。例如，

```
myList.add(2, d)
```

将 d 放在线性表的第二项——即在位置 2 处。但是，要这样做，需要将 b 移到位置 3，将 c 移到位置 4，所以线性表现在含有

```
a d b c
```

如果我们写下面的语句将 e 放到线性表的开头

```
myList.add(1, e)
```

则线性表中的当前项都要移向更高的一个位置。故线性表如下

```
e a d b c
```

再次看图 10-2，会明白这些操作的效果。

10.5 下面的语句可以得到这个线性表的第二项

```
entry2 = myList.getEntry(2)
```

现在变量 entry2 指向对象 a。记住，这里写的是伪代码，忽略了诸如分号这样的细节。

10.6 当删除一项时会发生什么？例如

```
myList.remove(3)
```

将从线性表中删除第三项——在前一个例子中是 d。然后线性表含有

```
e a b c
```

注意到，位于被删除项后面的项，都要向线性表中的下一个低位置移动。图 10-2 说明了线性表的这个变化。

如果应用程序需要我们从线性表中删除一项，但要保留这个项用作他用该怎么办呢？我们对 remove 的规范说明使得我们必须先使用 getEntry 来获取这个项，然后再使用 remove 从线性表中删除它。我们可以重新定义 remove 的规范说明，返回从线性表中删除的对象。要使用这个修改后的 remove 版本，可以写如下的伪代码语句：

```
oldEntry3 = myList.remove(3)
```

这样修改后使得 `remove` 的用途更广泛，而客户可以保存或忽略返回的项。

下面的语句可以用 `f` 替换线性表中的第三项 `b`

10.7

```
myList.replace(3, f)
```

其他的项都没有移动或改变。我们可以细化 `replace` 的规范说明，让它返回被替换的对象。这样，如果我们写

```
ref = myList.replace(3, f)
```

则 `ref` 应该指向之前的项 `b`。

 注：ADT 线性表中的对象具有由线性表的客户指定的次序。要添加、删除、获取或是替换一项，必须说明项在线性表中的位置。线性表中的第一个项在位置 1 处。

10.8

前面的规范说明和示例忽略了使用 ADT 线性表时可能出现的某些困难：

- 当给定的位置对于当前线性表无效时，操作 `add`、`remove`、`replace` 和 `getEntry` 应该有正确的动作。当这些操作接收一个无效的位置时将发生什么？
- 方法 `remove`、`replace` 和 `getEntry` 对空表是无意义的。对空表执行这些操作时将发生什么？

通常，我们必须决定如何处理这些状况，并细化我们的规范说明。ADT 线性表的文档应该反映前面的例子中说明的这些决策及细节。

我们重申第 1 章给出的下列说明，以提示大家。

 注：ADT 规范说明的初稿常常忽视或忽略你确实需要考虑的情形。你可能为了简化初稿而有意忽略这些。一旦写好了规范说明中的大部分内容，就可以关注于这些细节，而让规范说明更完善。

10.9

根据我们之前的讨论，现在来概括并改进 ADT 线性表的描述。

#### 抽象数据类型：线性表

##### 数据

- 按特定次序排列并有相同数据类型的对象的集合
- 集合中对象的个数

##### 操作

| 伪代码                                     | UML                                                        | 描述                                                                                                                                                                                                         |
|-----------------------------------------|------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>add(newEntry)</code>              | <code>+add(newEntry: T): void</code>                       | 任务：将 <code>newEntry</code> 添加到线性表尾<br>输入： <code>newEntry</code> 是一个对象<br>输出：无                                                                                                                              |
| <code>add(newPosition, newEntry)</code> | <code>+add(newPosition: integer, newEntry: T): void</code> | 任务：将 <code>newEntry</code> 添加到线性表的 <code>newPosition</code> 处。位置 1 表示线性表的第一项<br>输入： <code>newPosition</code> 是一个整数， <code>newEntry</code> 是一个对象<br>输出：如果在操作前 <code>newPosition</code> 是这个线性表中的无效位置，则抛出一个异常 |

(续)

| 伪代码                              | UML                                              | 描述                                                                                                                                                                         |
|----------------------------------|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| remove(givenPosition)            | +remove(givenPosition: integer): T               | 任务：删除并返回 givenPosition 位置的项<br>输入：givenPosition 是一个整数<br>输出：或者返回被删除的项，或者如果 givenPosition 是线性表中的无效位置时则抛出一个异常。注意，操作前如果线性表是空的，则任何的 givenPosition 值都是无效的                       |
| clear()                          | +clear(): void                                   | 任务：从线性表中删除所有的项<br>输入：无<br>输出：无                                                                                                                                             |
| replace(givenPosition, newEntry) | +replace(givenPosition: integer, newEntry: T): T | 任务：用 newEntry 替换 givenPosition 位置的项<br>输入：givenPosition 是一个整数，newEntry 是一个对象<br>输出：或者返回被替换的项，或者如果 givenPosition 是线性表中无效位置，则抛出一个异常。注意，操作前如果线性表是空的，则任何的 givenPosition 值都是无效的 |
| getEntry(givenPosition)          | +getEntry(givenPosition: integer): T             | 任务：获取 givenPosition 位置的项<br>输入：givenPosition 是一个整数<br>输出：或者返回 givenPosition 位置的项，或者如果 givenPosition 是线性表中无效位置，则抛出一个异常。注意，操作前如果线性表是空的，则任何的 givenPosition 值都是无效的             |
| toArray()                        | +toArray: T[]                                    | 任务：按项在线性表中出现的次序获取表中的所有项<br>输入：无<br>输出：返回含有线性表中当前项的新数组                                                                                                                      |
| contains(anEntry)                | +contains(anEntry: T): boolean                   | 任务：查看线性表中是否含有 anEntry<br>输入：anEntry 是一个对象<br>输出：如果 anEntry 在线性表中，则返回真，否则返回假                                                                                                |
| getLength()                      | +getLength(): integer                            | 任务：得到线性表中当前项的个数<br>输入：无<br>输出：返回线性表中当前项的个数                                                                                                                                 |
| isEmpty()                        | +isEmpty(): boolean                              | 任务：检查线性表是否为空<br>输入：无<br>输出：如果线性表为空，则返回真，否则返回假                                                                                                                              |

10.10 程序清单 10-1 中的 Java 接口含有用于 ADT 线性表的方法及描述其行为的详细注释。这些注释细化了前一段给出的规范说明。线性表中的项是同一个类或有继承关系的类的对象。

### 程序清单 10-1 接口 ListInterface

```

1 /** An interface for the ADT list.
2 Entries in a list have positions that begin with 1.
3 */
4 public interface ListInterface<T>
5 {
6 /** Adds a new entry to the end of this list.
7 Entries currently in the list are unaffected.
8 The list's size is increased by 1.
9 @param newEntry The object to be added as a new entry. */
10 public void add(T newEntry);
11
12 /** Adds a new entry at a specified position within this list.
13 Entries originally at and above the specified position
14 are at the next higher position within the list.
15 The list's size is increased by 1.
16 @param newPosition An integer that specifies the desired
17 position of the new entry.
18 @param newEntry The object to be added as a new entry.
19 @throws IndexOutOfBoundsException if either
20 newPosition < 1 or newPosition > getLength() + 1. */
21 public void add(int newPosition, T newEntry);
22
23 /** Removes the entry at a given position from this list.
24 Entries originally at positions higher than the given
25 position are at the next lower position within the list,
26 and the list's size is decreased by 1.
27 @param givenPosition An integer that indicates the position of
28 the entry to be removed.
29 @return A reference to the removed entry.
30 @throws IndexOutOfBoundsException if either
31 givenPosition < 1 or givenPosition > getLength(). */
32 public T remove(int givenPosition);
33
34 /** Removes all entries from this list. */
35 public void clear();
36
37 /** Replaces the entry at a given position in this list.
38 @param givenPosition An integer that indicates the position of
39 the entry to be replaced.
40 @param newEntry The object that will replace the entry at the
41 position givenPosition.
42 @return The original entry that was replaced.
43 @throws IndexOutOfBoundsException if either
44 givenPosition < 1 or givenPosition > getLength(). */
45 public T replace(int givenPosition, T newEntry);
46
47 /** Retrieves the entry at a given position in this list.
48 @param givenPosition An integer that indicates the position of
49 the desired entry.
50 @return A reference to the indicated entry.
51 @throws IndexOutOfBoundsException if either
52 givenPosition < 1 or givenPosition > getLength(). */
53 public T getEntry(int givenPosition);
54
55 /** Retrieves all entries that are in this list in the order in which
56 they occur in the list.
57 @return A newly allocated array of all the entries in the list.
58 If the list is empty, the returned array is empty. */
59 public T[] toArray();
60
61 /** Sees whether this list contains a given entry.
62 @param anEntry The object that is the desired entry.
63 @return True if the list contains anEntry, or false if not. */

```

```

64 public boolean contains(T anEntry);
65
66 /** Gets the length of this list.
67 @return The integer number of entries currently in the list. */
68 public int getLength();
69
70 /** Sees whether this list is empty.
71 @return True if the list is empty, or false if not. */
72 public boolean isEmpty();
73 } // end ListInterface

```

### 设计决策：当集合为空时，应该由谁决定 toArray 做什么？

方法 `toArray` 返回新分配的含有集合中项的数组。当集合为空时，方法可以

- 返回一个空数组，或是
- 抛出一个异常

当哪个行为都合情合理时，是应该让 ADT 的设计者做出选择，并在接口中记录下来？还是设计者忽略这种情形而让接口的实现者来做选择？

例如，当我们写程序清单 10-1 中的 `ListInterface` 时，我们决定，如果线性表为空，则 `toArray` 应该返回一个空数组。所以，任何实现 `ListInterface` 的类都必须遵从这个决策。而且，这样一个类的所有客户都可以期待 `toArray` 以这种方式运行。现在假定，`ListInterface` 中的 `toArray` 没有提及当线性表为空时将会发生什么。写实现 `ListInterface` 的类的程序员可以决定那种情形下 `toArray` 将做什么。这样的实现可以各不相同。例如，类 `XList` 中的 `toArray` 方法可能会抛出一个异常，而 `YList` 中的 `toArray` 方法可能会返回一个空数组。哪怕我们明确说明这些类中 `toArray` 的行为，客户仍不能在不改变 `toArray` 的使用的情况下用一个类来替代另一个类。

我们的决策是，完整地给出接口中每个方法的规范说明，以便它的不同实现都可以产生相同的行为。

 **学习问题 1** 写伪代码语句，按照下面的要求将一些对象添加到线性表中。先添加 c，然后是 a，再添加 b，再添加 d，这些操作后，线性表中对象的次序将是 a,b,c,d。

**学习问题 2** 写伪代码语句，交换含 10 个对象的线性表中的第 3 项与第 7 项。

 注：含  $n$  项的线性表中的项按从 1 到  $n$  进行编号。虽然你不能在位置 0 处添加新项，但可以在位置  $n+1$  处添加。

## 使用 ADT 线性表

想象一下，我们雇一位程序员使用 Java 实现 ADT 线性表，给定目前已有的接口和规范说明。如果我们假定，这些规范说明足够让程序员完成相应的实现任务，那么我们就可以在程序中使用 ADT 的操作而无须了解实现的细节。即我们不必知道程序员是如何实现的线性表，也能够使用它。我们仅需知道 ADT 线性表做什么就可以了。

本节假定，我们实现了线性表，并说明在程序中如何使用线性表。这里的示例可作为用来测试你的实现的程序的一部分。

示例。想象一下，我们正在组织一个本地公路赛。我们的任务是记下赛跑选手们完成比赛的次序。因为每位选手都佩戴一个独特的识别号，所以当每位选手冲过终点线时我们可以将他的号码加到线性表尾。图 10-3 说明了这样的一个线性表。



图 10-3 按完成比赛的次序排列的运动员识别号码的线性表

程序清单 10-2 中的 Java 程序显示，我们如何使用 ADT 线性表来完成这个任务。假定类 AList 实现了前一节看到的 Java 接口 ListInterface。因为 ListInterface 假定，线性表中的项都是对象，所以我们将每位选手的识别号看作一个字符串。

### 程序清单 10-2 实现 ListInterface 的类的客户

```

1 public class RoadRace
2 {
3 public static void main(String[] args)
4 {
5 recordWinners();
6 } // end main
7
8 public static void recordWinners()
9 {
10 ListInterface<String> runnerList = new AList<>();
11 // runnerList has only methods in ListInterface
12
13 runnerList.add("16"); // Winner
14 runnerList.add("4"); // Second place
15 runnerList.add("33"); // Third place
16 runnerList.add("27"); // Fourth place
17 displayList(runnerList);
18 } // end recordWinners
19
20 public static void displayList(ListInterface<String> list)
21 {
22 int numberofEntries = list.getLength();
23 System.out.println("The list contains " + numberofEntries +
24 " entries, as follows:");
25
26 for (int position = 1; position <= numberofEntries; position++)
27 System.out.println(list.getEntry(position) +
28 " is entry " + position);
29
30 System.out.println();
31 } // end displayList
32 } // end RoadRace

```

**输出**

```
The list contains 4 entries, as follows:
16 is entry 1
4 is entry 2
33 is entry 3
27 is entry 4
```

`displayList` 的输入参数 `list` 的数据类型是 `ListInterface<String>`。所以，方法的参数必须是同时满足下列两个条件的一个对象：

- 对象的类必须实现了 `ListInterface`。
- 对象必须是字符串线性表的实例。

尽管方法适用于 ADT 线性表的任何实现，但它仅能用于字符串线性表。如下修改方法头就可以去掉后一条限制：

```
public static <T> void displayList(ListInterface<T> list)
```

现在，传给方法的线性表可以含有任何类的对象。

**注：提醒**

注意到，`runnerList` 的数据类型是 `ListInterface<String>`。这个声明要求 `runnerList` 仅能调用接口中的方法，且仅能将字符串添加到线性表中。如果数据类型换为 `AList<String>`，则 `runnerList` 应该能调用 `AList` 中的任何公有方法，哪怕它们没有在 `ListInterface` 中声明。



**学习问题 3** 在前一个例子中，要想将选手的号码表示为 `Integer` 对象而不是字符串，则方法 `recordWinners` 必须做哪些改变？使用在线的补充材料 1 的段 S1.99 中所描述的 Java 自动装箱功能实现。

10.12



**示例。**一位教授想要一份按字典序排列的今天来上课的学生名单。每位学生进入教室，教授将学生的名字添加到线性表中。根据教授的意愿，将每个名字放到线性表中的正确位置，以便名字将按字典序排列。ADT 线性表不能选择项的次序。

下列 Java 语句将 Amy、Elias、Bob、Drew、Aaron 和 Carol 放到按字典序排列的线性表中。每个语句后面的注释列出语句执行后线性表中的情形。

```
// Make an alphabetical list of names as students enter a room
ListInterface<String> alphaList = new AList<>();
alphaList.add(1, "Amy"); // Amy
alphaList.add(2, "Elias"); // Amy Elias
alphaList.add(2, "Bob"); // Amy Bob Elias
alphaList.add(3, "Drew"); // Amy Bob Drew Elias
alphaList.add(1, "Aaron"); // Aaron Amy Bob Drew Elias
alphaList.add(4, "Carol"); // Aaron Amy Bob Carol Drew Elias
```

最先将 Amy 添加到线性表的开头后，Elias 添加到线性表尾（在位置 2 处），教授插入

- Bob 在 Amy 和 Elias 的中间，位于位置 2 处
- Drew 在 Bob 和 Elias 的中间，位于位置 3 处
- Aaron 在 Amy 的前面，位于位置 1 处
- Carol 在 Bob 和 Drew 的中间，位于位置 4 处

你将在第 15 章学习到，将每个名字插入字典序排列的名字集合中的这项技术称为插入排序。

如果现在写下面的语句，则删除位置 4 的项——Carol

```
alphaList.remove(4);
```

则 Drew 和 Elias 分别处于位置 4 和位置 5。所以 `alphaList.getEntry(4)` 将返回指向 Drew 的引用。

最后，假定我们想替换这个线性表中的一个名字。不能用随意的名字来替换，除非线性表仍维持字典序。写用 Ben 替换 Bob 的语句：

```
alphaList.replace(3, "Ben");
```

将保持字典序，但用 Nancy 替换 Bob 时则不能保持。线性表的字典序是由我们最初制订的策略所决定的，这个策略决定将名字添加在线性表的何处。这个次序不是线性表操作的自然结果。即是客户而不是线性表来维护这个次序。但我们可以设计一个 ADT，它能按字典序维护数据。在第 17 章会看到这样的 ADT 的示例。

 **学习问题 4** 假定 `alphaList` 是含有 Amy、Elias、Bob 和 Drew 共 4 个字符串名字的线性表。写交换 Elias 和 Bob，然后交换 Elias 和 Drew 的 Java 语句，让线性表仍保持字典序。

 **示例**。现在来看看不是字符串对象的线性表。假定我们有附录 B 程序清单 B-1 中表示人的姓名的类 `Name`。下列语句表明我们如何得到由 Amy Smith、Tina Drexel 和 Robert Jones 组成的线性表：

```
// Make a list of names as you think of them
ListInterface<Name> nameList = new AList<>();
Name amy = new Name("Amy", "Smith");
nameList.add(amy);
nameList.add(new Name("Tina", "Drexel"));
nameList.add(new Name("Robert", "Jones"));
```

现在，获取线性表中处在第二位的名字 Tina Drexel：

```
Name secondName = nameList.getEntry(2);
```

`getEntry` 的定义声明了它的返回值类型是 `T`，这是线性表中项的泛型。这个类型对于 `nameList` 来说是 `Name`，所以 `getEntry` 返回一个 `Name` 对象。

 **学习问题 5** 假定 `getEntry` 的返回值类型是 `Object`，而不是泛型。这个改变对这个方法的使用有什么影响吗？具体来说，前一个例子中获取 `nameList` 中第二个名字的语句还正确吗？为什么？

 **示例**。现在对前一个示例再多说两句。变量 `secondName` 是一个引用，它指向线性表中的第二个对象。使用这个引用可以修改对象。例如，下面的语句可以修改最后一个名字：

```
secondName.setLast("Doe");
```

如果类 `Name` 没有像 `setLast` 这样的设置方法，则我们不能修改这个线性表中的对

10.13

10.14

象。例如，如果我们有一个字符串表，则不能通过这种方式修改其中任何一个字符串。类 `String` 没有设置方法，所以一旦创建了一个 `String` 对象，我们就不能改变它。但我们可以用 ADT 线性表操作 `replace`，替换线性表中的整个对象——不管它是什么类型的。

回忆第 7 章提到过的可变对象和不可变对象。因为类 `Name` 有设置方法，所以它的对象是可变的。而另一方面，类 `String` 没有定义设置方法，所以它的对象是不可变的。

## 问题求解：使用大整数



在 Java 中可以表示相当大的整数：`int` 类型的最大正整数是 2 147 483 647，而 `long` 类型的是 9 223 372 036 854 775 807。不过，如果我们测量天体之间的距离，或是计算沙粒数，则前面的 10 位和 19 位整数都太小了。虽然 Java 的类 `BigInteger` 提供了 2 的补码记号下任意精度的整数，且 `BigDecimal` 表示任意精度的十进制实数，但我们想要一个简单的十进制整数类。让我们设计自己的非常大的基数是 10 的整数类 `BigInt`。

10.15

我们先从规范说明类 `BigInt` 中的一些方法开始。

`+set(String value): void`

任务：使用 `value` 中的数字串设置本 `BigInt` 的值。

`+setToZero(): void`

任务：将本 `BigInt` 设置为 0。

`+getLength(): integer`

任务：返回本 `BigInt` 值中的数字个数。

`+add(BigInt operand2): BigInt`

任务：将 `this` 与 `operand2` 的和作为 `BigInt` 返回。

`+subtract(BigInt operand2): BigInt`

任务：将 `this` 与 `operand2` 的差作为 `BigInt` 返回。

`+multiply(BigInt operand2): BigInt`

任务：将 `this` 与 `operand2` 的乘积作为 `BigInt` 返回。

`+divide(BigInt operand2): BigInt`

任务：将 `this` 与 `operand2` 的商作为 `BigInt` 返回。

`+compareTo(BigInt operand): integer`

任务：比较 `this` 和 `operand`；如果 `this` 小于 `operand` 则返回一个负整数，如果两者相等则返回 0，如果 `this` 大于 `operand` 则返回一个正整数。

`+equals(BigInt operand): boolean`

任务：比较 `this` 和 `operand`，根据两值是否相等返回真或假。

`+toString(): String`

任务：将本 `BigInt` 按十进制值作为字符串返回。

当  $n$  可能非常大时我们应该如何表示一个  $n$  位十进制整数？实际上，因为计算机的局限性我们只能限制  $n$  的大小。有时，大整数要输入程序中。不论它是从键盘读入的，还是从一个文件中读入的，或者是 Java 代码中的文本值，每个整数都是一个字符串。我们可以如下这样处理：

- 将字符串按原样保留并按需操作以执行大整数运算。
- 从最右边的数字开始将字符串按 9 位一组进行划分。可以将每 9 个数字表示为一个 `int` 型值，并将这些整数值放到一个线性表中。
- 从字符串中提取每位数字，并按字符、字符串或是整数形式放到线性表中。

10.16

我们选择最后这种方式。数字应该位于线性表中的什么位置？例如，要用线性表表示整数 1234，第一个数字 1 应该位于线性表中的第一个、最后一个还是中间结点？因为 ADT 线性表在理论上没有结尾，所以我们只能确定它的第一个位置。让我们来修改选择。应该将整数的首位，即最高有效位数字放在线性表的第一个位置吗？或者应该将整数最后一位，即最低有效位数字放在线性表的第一个位置？你可能知道，1234 是计算  $4 \times 10^0 + 3 \times 10^1 + 2 \times 10^2 + 1 \times 10^3$  得到的值。另外，因为 `BigInt` 的算术运算要求我们从最低有效数字开始，所以将 4 放到线性表的第一个位置，将 3 放到线性表的第二个位置，等等。这种情形下线性表是这个样子的：

```
4
3
2
1
```

10.17

我们来看看如何将两个 `BigInt` 值相加。考虑下列加法：

$$\begin{array}{r} 1234 \\ +5678 \\ \hline \end{array}$$

10.18

应该相加最右面的数字，4 和 8，得到 12。2 是两个 `BigInt` 值之和值的最右面的数字，而 1 要进位到下一对数字 3 和 7 的相加中。当得到最终和值的数字时，应该将它们保存在线性表中。



### 设计决策：应该如何在线性表中表示数字？

十进制整数中的数字是 0 ~ 9。应该将每个数字保存为一个字符串、一个字符、一个整数还是其他的数据类型？将数字表示为一个 `String` 对象，表示为 `char` 或 `Character` 类型的一个 `Unicode` 字符，或是表示为 `int` 或 `Integer` 类型的一个整数，都需要明显多得多的存储空间。因为一个 `BigInt` 值可能含有非常非常多的数字，所以我们应该节约地使用空间。将每个数字表示为 `byte` 或 `Byte` 值会节省空间。因为我们将数字保存在线性表中，需要将数字作为对象使用，故使用包装类 `Byte`。但类 `Byte` 没有提供算术运算，所以我们需要将每个数字转为一个整数。实际上，类 `Byte` 中提供了一个方法，可以将一个字节转为一个整数。利用这个便利并在线性表外执行算术运算，故我们用到的存储很小。为将一个整数转为一个字节以便可以将它放到线性表中，可以使用类 `Integer` 中的方法。

本章最后的项目 13 要求你完成类 `BigInt` 的设计和定义。

## Java 类库：接口 List

**10.19** 标准包 `java.util` 中含有接口 `List`，它用于 ADT 线性表，类似于我们的接口中描述的线性表。Java 类库中的线性表和我们的 ADT 线性表的一个不同之处在于，对表中项的编号不同。在 Java 类库中的线性表，使用与 Java 数组中一样的编号机制：第一项在位置或下标 0 处。而我们的线性表从位置 1 开始。另外，接口 `List` 比我们的接口声明了更多的方法。在 Java 插曲 4 中你会看到几个额外的方法。

从接口 `List` 中选择了以下几个方法头，它们类似于你在本章看到的方法。与我们方法的不同之处已做了标记。再次说明，`T` 是线性表中项的泛型。

```
public boolean add(T newEntry)
public void add(int index, T newEntry)
public T remove(int index)
public void clear()
public T set(int index, T anEntry) // Like replace
public T get(int index) // Like getEntry
public boolean contains(Object anEntry)
public int size() // Like getLength
public boolean isEmpty()
```

第一个 `add` 方法，将一个项添加到线性表尾，返回一个布尔值，而我们类似的方法是一个 `void` 方法。方法 `set` 类似于我们的 `replace` 方法，而方法 `get` 类似于我们的 `getEntry` 方法。`contains` 参数的数据类型是 `Object`，而不是泛型。实际上，这个差别并不重要。最后，方法 `size` 类似于我们的 `getLength` 方法。

可以参考 Java 类库的在线文档，更详细了解接口 `List`。

## Java 类库：类 ArrayList

**10.20** Java 类库中含有一个使用可变大小的数组来实现的 ADT 线性表。称为 `ArrayList` 的这个类实现了我们刚讨论的接口 `java.util.List`。这个类也在包 `java.util` 中。

`ArrayList` 类的两个构造方法如下：

```
public ArrayList()
```

创建一个初始容量为 10 的空线性表。表按需增大容量，数目不定。

```
public ArrayList(int initialCapacity)
```

创建有指定初始容量的空线性表。表按需增大容量，数目不定。

**10.21** 第 6 章描述过的类 `java.util.Vector` 类似于 `ArrayList`。两个类都实现了相同的接口：`java.util.List` 及其他接口。即便如此，`Vector` 中所含的方法比 `ArrayList` 多几个。我们忽略这些额外的方法，因为它们大多数都是冗余的。

可以使用 `ArrayList` 或是 `Vector` 作为接口 `List` 的实现。例如，下面两个语句都可用来定义字符串线性表：

```
List<String> myList = new ArrayList<>();
```

或

```
List<String> myList = new Vector<>();
```

现在 `myList` 仅有声明在接口 `List` 中的方法。

我们的 `ListInterface` 比 Java 的 `List` 要简单一些，因为它含有更少的方法。可以让我们的接口保持简单，同时通过使用实现 `ListInterface` 的 `ArrayList` 或是 `Vector` 来利用已有的类。这种方式类似于用在第 6 章程序清单 6-3 中实现 ADT 栈时的做法。将细节留作程序设计项目。

## 本章小结

- 线性表是一个对象，其数据由有次序的项构成。每个项由其在线性表中的位置来标识。
- ADT 线性表规范说明了将一个项添加在线性表尾或是线性表中指定位置的操作。其他的操作是获取、删除或是替换指定位置的项。
- 客户仅能使用 ADT 线性表中定义的操作来处理或访问线性表中的项。
- 包中的项是无序的，而线性表、栈、队列、双端队列或是优先队列中的项是有次序的。线性表，与上述其他集合不同，能让你添加、获取、删除或是替换任意给定位置的项。

## 练习

1. 如果 `myList` 是一个空的字符串线性表，执行下列语句后所含的内容是什么？

```
myList.add("A");
myList.add("B");
myList.add("C");
myList.add("D");
myList.add(1, "one");
myList.add(1, "two");
myList.add(1, "three");
myList.add(1, "four");
```

2. 如果 `myList` 是一个空的字符串线性表，执行下列语句后所含的内容是什么？

```
myList.add("alpha");
myList.add(1, "beta");
myList.add("gamma");
myList.add(2, "delta");
myList.add(4, "alpha");
myList.remove(2);
myList.remove(2);
myList.replace(3, "delta");
```

3. 修改程序清单 10-2 中的方法 `displayList`，让它使用线性表的方法 `toArray` 而不是用 `getLength` 和 `getEntry`。

4. 假定你需要 ADT 线性表中的一个操作，它返回给定对象在线性表中的位置。方法头可能如下所示：

```
public int getPosition(T anObject)
```

其中 `T` 是线性表中对象的泛型。写出注释，规范说明这个方法。

5. 假定你需要 ADT 线性表中的一个操作，它删除给定对象在线性表中的第一次出现。方法头可能如下所示：

```
public boolean remove(T anObject)
```

其中 `T` 是线性表中对象的泛型。写出注释，规范说明这个方法。

6. 假定你需要 ADT 线性表中的一个操作，它将线性表的第一项移到线性表的最后。方法头可能如下

所示：

```
public void moveToEnd()
```

写出注释，规范说明这个方法。

7. 写出客户层的 Java 语句，它返回给定对象在线性表 myList 中的位置。假定对象位于线性表中。
8. 假定 ADT 线性表中没有方法 replace。写出客户层的 Java 语句，替换线性表 nameList 中的一个对象。对象在线性表中的位置是 givenPosition，替换对象是 newObject。
9. 假定 ADT 线性表中没有方法 contains。再假定 nameList 是 Name 对象的线性表，其中 Name 如附录 B 的程序清单 B-1 中的定义。写出客户层的 Java 语句，判定 Name 对象 myName 是否在线性表 nameList 中。
10. 假定有由下列语句创建的线性表：

```
ListInterface<Student> studentList = new AList<>();
```

假定某人已经向表中添加了附录 C 的段 C.2 定义的类 Student 的几个实例。写出客户层的 Java 语句，完成下列任务

- a. 按学生在线性表中出现的次序显示学生的姓。不改变线性表。
- b. 交换线性表中第一个和最后一个学生。

11. 假定有由下列语句创建的线性表：

```
ListInterface<Double> quizScores = new AList<>();
```

假定某人已经向线性表中添加了一位学生在整个课程中得到的测验成绩。教授想知道这些测验成绩的平均分，忽略最低成绩。写出客户层的 Java 语句，完成下列任务

- a. 找到并删除线性表中的最低分。
- b. 计算线性表中剩余成绩的平均值。

12. 考虑表示硬币的类 Coin。类有方法 getValue、toss 和 isHeads。方法 getValue 返回硬币的面值或面额。方法 toss 模拟一次硬币抛掷，抛掷后硬币落地时或者正面朝上或者背面朝上。如果正面朝上，则方法 isHeads 返回真。

假定 coinList 是一个 ADT 线性表，含有随机选择面额的硬币。抛掷每个硬币。如果硬币抛掷的结果是正面朝上，则将硬币移到称为 headsList 的第二个线性表中；如果它是背面朝上，则将它留在原线性表中。当完成抛掷时，计算所有正面朝上的硬币的总面值。假定线性表 headsList 已经创建好且初始为空。

## 项目

1. 使用标准类 Vector，定义线性表类 OurList，其实现了本章程序清单 10-1 中定义的接口 ListInterface。
2. 重复前一个项目，但使用类 ArrayList 而不是使用 Vector。  
在下列项目中当你需要使用线性表时，使用项目 1 或项目 2 要求你定义的 OurList 类。
3. 定义一个包的类，其实现了第 1 章程序清单 1-1 中定义的接口 BagInterface。使用类 ArrayList 的实例来保存包的项。然后写一个程序，充分论证你的新类。注意，可能必须处理 ArrayList 方法抛出的异常。
4. 重做项目 3，但要定义集合的类，其实现了第 1 章程序清单 1-5 中定义的接口 SetInterface。回忆一下，集合是其中的项互不相同的一个包。
5. 重做项目 3，但要定义栈的类，它实现了第 5 章程序清单 5-1 中定义的接口 StackInterface。
6. 重做项目 3，但要定义队列的类，它实现了第 7 章程序清单 7-1 中定义的接口 QueueInterface。

7. 重做项目 3，但要定义双端队列的类，它实现了第 7 章程序清单 7-4 中定义的接口 `DequeInterface`。
8. 据称圣诞老人克劳斯有淘气孩子和好孩子名单。淘气孩子名单上的人的袜子里有煤。好孩子名单上的人会有礼物。这个名单中的每个对象都含有一个名字（附录 B 程序清单 B-1 定义的 `Name` 的实例）和这个人的礼物清单（ADT 线性表的一个实例）。

设计好孩子名单中的人的 ADT。规范说明每个 ADT 操作，说明其目的，描述它的形参，写前置条件及后置条件，给出方法头的伪代码。写一个包含 `javadoc` 风格注释的用于 ADT 的 Java 接口。然后使用 `ArrayList` 的实例保存礼物清单实现这个 ADT。

设计并实现类 `SantaClaus`，维护淘气孩子和好孩子名单。写程序论证这个类。

9. 食谱含有一个名称、一个配料表和一组操作指南。配料表中的项含有数量、单位和描述。例如，这个表中的一个项可能是表示 2 杯面粉的一个对象。操作指南表中的项是一个字符串。

设计一个 ADT，表示配料表中的一项，假定你已有类 `MixedNumber`，定义在序言的项目 6 中。然后设计另一个 ADT，表示任何食谱。规范说明每个 ADT 操作，说明其目的，描述它的形参，写前置条件及后置条件，给出方法头的伪代码。然后写一个包含 `javadoc` 风格注释的用于 ADT 的 Java 接口。

10. 定义并测试实现前一个项目中描述的 ADT 食谱接口的一个类。将接口 `ArrayList` 的实例用于你需要的每个线性表。使用文本编辑器创建一个文本文件，作为示例程序必须读入的食谱。

11. 重做第 4 章项目 7，使用 `ArrayList` 的实例而不用包。

12. 早在 10 世纪，数学家研究了下列整数的三角图案，现在称为帕斯卡三角（中国称为杨辉三角——译者注）：

```

 1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1
 ...

```

虽然这个图案出现的很早，但它是以 17 世纪数学家布莱士·帕斯卡的名字命名的。

按惯例，错开排列项，如这里所示的。每一行的开始和结束都是 1。每个内部的项都是它上面两项的和。例如，这里给出的最后一行中，4 是上面 1 和 3 的和，6 是 3 和 3 的和，而 4 是 3 和 1 的和。

如果我们都从 0 开始对行和每行中的项进行编号，则行  $n$  中位置  $k$  的项常表示为  $C(n, k)$ 。例如，最后一行中 6 是  $C(4, 2)$ 。给定  $n$  个项， $C(n, k)$  的结果是在  $n$  个项中选择  $k$  个项的不同选法。所以， $C(4, 2)$  是 6，从给定的 4 个项中选择 2 个，有 6 种选法。所以，如果 A、B、C 和 D 是这 4 个项，下面是 6 种可能的选法：

AB, AC, AD, BC, BD, CD

注意到，每一对中项的次序是无关的。实际上，选项 AB 与选项 BA 是一样的。

设计并实现类 `PascalTriangle`。将三角中的一行表示在一个线性表中，整个三角表示为这些线性表的线性表。使用 `ArrayList` 类表示这些线性表。在类中要包括构造方法，且至少要有方法 `getChoices(n, k)`，它返回  $C(n, k)$  的整数值。

13. 考虑段 10.15 中介绍的 `BigInt` 类，及项目 1 中所写的 `ListInterface` 和 `OurList` 类。为 `BigInt` 定义一个接口 `BigIntInterface`，从而完成它的设计。然后完成 `BigInt` 的定义。

14. (游戏) 考虑赛马的起跑门。假定  $n$  匹马进入赛道，被指派到  $1 \sim n$  的起点位置处。如果有一匹马未能进入起跑门并被取消资格，其他的马匹不改变位置。

- a. 设计一个 ADT 表示起跑门。

- b. 使用 ADT 线性表的操作，写伪代码实现你的 ADT。
  - c. 使用项目 1 和项目 2 要求你完成的 OurList 类实现你的 ADT。
15. (财务) 支票簿上的支票有连续的编号，可以从任意给定的正整数开始。
- a. 设计 ADT 支票及 ADT 支票簿。
  - b. 使用 ADT 线性表的操作，写伪代码实现你的 ADT。
  - c. 使用项目 1 和项目 2 要求你完成的 OurList 类实现你的 ADT。
16. (电子商务) 有些网上商店为顾客提供礼物登记处，顾客可以在此输入他们希望收到的作为特殊事件的礼物。我们在登记表中让顾客按喜爱的优先顺序排列礼物。
- a. 设计一个表示礼物登记处的 ADT。
  - b. 使用 ADT 线性表的操作，写伪代码实现你的 ADT。
  - c. 使用项目 1 和项目 2 要求你完成的 OurList 类实现你的 ADT。
17. 编译程序必须检查程序中的记号，以确定它们是否是保留字或是用户定义的标识符。设计一个程序，读入一段 Java 程序，生成一个线性表用来保存用户定义的所有标识符。你需要第二个线性表用来保存 Java 的所有保留字。当遇到一个记号时，查找保留字线性表。如果记号不是保留字，则再查找用户定义的标识符线性表。如果记号在两个线性表中都没有出现，则应该将它添加到标识符线性表中。

# 使用数组实现线性表

先修章节：序言、第 2 章、第 4 章、第 10 章

## 目标

学习完本章后，应该能够

- 使用变长数组实现 ADT 线性表
- 讨论给出的实现方案的优缺点

你已经见过了几个示例，示范在程序中如何使用 ADT 线性表。现在来讨论在 Java 中实现线性表的几种不同方法。本章使用数组来表示线性表中的项。正如之前其他 ADT 的实现一样，当用完了数组中的所有空间后，可以将数据移到一个更大的数组中。虽然我们将它留作一个项目，不过你仍可以使用 Java 的 `ArrayList` 或 `Vector` 类的实例来表示线性表的项。这个用法与使用可扩展的数组是一样的，因为 `ArrayList` 或 `Vector` 的底层实现使用的是这样一个数组。最后，下一章将线性表保存在结点链表中。因为可以在线性表的任何位置插入和删除项，故基于数组和链式的实现，比我们之前遇到的 ADT 的实现更具挑战性。

## 使用数组实现 ADT 线性表

考虑第 2 章讨论过的教室，我们用它来比喻一个数组如何表示一个包，不过这次我们展示如何表示一个线性表。为此目的，我们展示 `add` 和 `remove` 方法将如何工作。随后，使用 Java 实现线性表。

### 模拟

我们回忆第 2 章用过的教室房间 A。在图 11-1 中再次画出它。房间中的椅子从 0 开始顺序编号。数组很像这间教室，每把椅子很像数组中的一个元素。我们再次将教室看作一维数组，且忽略在一间典型的教室内那些椅子是按行排列的。11.1

假定我们想从 1 而不是从 0 对学生进行编号。可以忽略椅子 0，或让教师使用它。所以，到达教室的第一位学生坐在椅子 1 上；第二位学生坐在椅子 2 上，以此类推。最终，30 名学生占据了 1~30 号的椅子。他们是按到达的时间组织的。教师立即知道谁最先到达（坐在椅子 1 上的人），谁最后到达（坐在椅子 30 上的人）。另外，教师可以询问坐在某把椅子上的学生姓名，正如程序员可以直接访问任意的数组元素一样。所以，教师通过轮询从 1 号到 30 号椅子，就可以按到达次序查询每位学生的姓名，或是轮询从 30 号到 1 号椅子，按到达的相反次序询问。

我们可以不按学生到达房间 A 的时间来安排，而是按姓名的字典序来安排。为达此目的，需要一个排序算法，如将在第 15 章和第 16 章讨论的。也就是说，ADT 线性表不选择项的次序；而客户必须来选择。

按姓名的字典序添加一位新学生。想象一下，我们已经按姓名的字典序将学生安排在房间 A 中。假定一名新学生想加入已在房间的学生中。回忆一下，30 把已占用的椅子已经从11.2

1 到 30 顺序编号。因为房间中有 40 把椅子，所以可用编号为 31 的椅子。当学生按到达时间安排时，我们可以简单地将新学生安排坐在椅子 31 上。因为现在学生是按姓名的字典序安排的，所以我们必须做更多的工作。

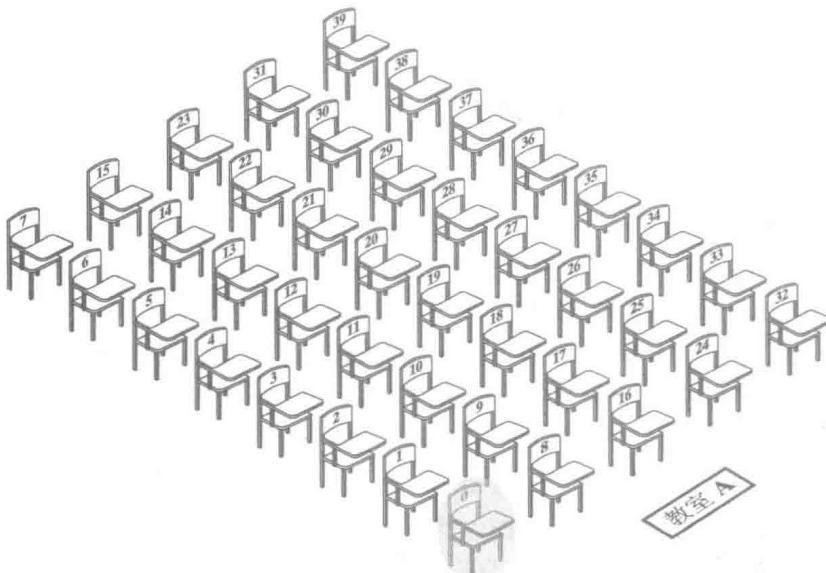


图 11-1 椅子在固定位置的教室

假定新学生介于坐在椅子 10 和椅子 11 上的两位学生之间。即新学生的名字按字典序位于坐在椅子 10 和椅子 11 上的两位学生的名字之间。因为椅子的位置是固定的，新学生必须坐在椅子 11 上。新学生坐下之前，目前坐在椅子 11 上的学生必须移到椅子 12 上，如图 11-2 所示。但这个要求，会引起一系列的反应：目前坐在椅子 12 上的学生必须移到椅子 13 上，以此类推。即坐在 11 ~ 30 号椅子上的每位学生都必须移到下一个更大一号的椅子上。如果一次只一位学生移动，则椅子 30 上的学生必须先移到椅子 31 上，然后椅子 29 上的学生才能移到椅子 30 上，以此类推。正如你看到的，增加一位新学生需要移动若干其他的学生。但是，我们不会打扰到坐在新学生的椅子之前的椅子——在本例中是 1 ~ 10 号——上的学生。



图 11-2 新学生坐在两位已坐下的学生中间：至少有一位学生必须移动



**学习问题 1** 在前一个例子中，什么情况下，你能按名字的字典序添加一位新学生但不移动任何其他学生？

删除一位学生。现在想象，房间 A 中椅子 5 上的学生要逃课。椅子放在房间内的固定位置。如果我们仍想让学生坐在连续编号的椅子上，则有几位学生必须要移动。实际上，坐在 6 号及大于 6 号椅子上的每位学生都必须移到更小一号的椅子上，从坐在椅子 6 上的学生开始。也就是说，如果一次只一位学生移动，则坐在椅子 6 上的学生必须先移到椅子 5 上，然后坐在椅子 7 上的学生才能移到椅子 6 上，以此类推。



**学习问题 2** 刚描述的为使腾空的椅子不空置而让学生移动的优点是什么？

**学习问题 3** 让腾空的椅子空置的优点是什么？

## Java 实现

Java 基于数组实现 ADT 线性表的方案中，包含了我们在教室示例图中的一些思想。我们的实现是类 `AList`<sup>②</sup>，它实现了第 10 章见过的 `ListInterface` 接口。类内的每个公有方法对应一个 ADT 线性表操作。私有数据域是

- 一个对象数组
- 记录线性表中项的个数的整数
- 定义线性表默认容量的整型常数
- 定义线性表最大容量的整型常数
- 表示线性表是否已正确初始化的布尔标志

可以使用图 11-3 中所示的 UML 符号来描述类，其中 T 表示线性表中项的数据类型。我们已经添加了数据域 `MAX_CAPACITY` 和 `integrityOK`，它们有助于提升类的安全性，正如我们在前几章其他类中所做的一样。

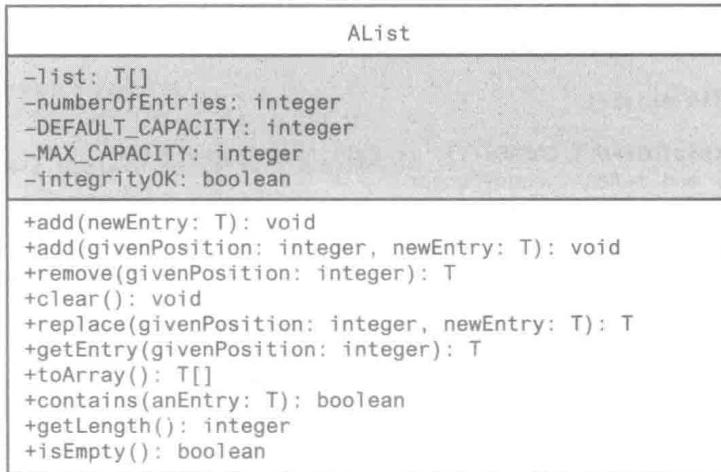


图 11-3 用于类 `AList` 的 UML 符号

② 按说，我们应该将这个类称为 `ArrayList`。但正如你在第 10 章看到的，Java 已经提供了有这个名字的类。虽然我们肯定能将自己的类命名为 `ArrayList`，不过我们还是选择不同的名字以避免冲突。



### 设计决策：数组中的哪些元素应该用来保存线性表？

在第 10 章中，我们凭直觉将线性表中的项进行了编号，即第一个项在位置 1。数组中的哪些元素应该用来保存线性表？我们应该将第一个项放到 `list[0]` 中，这样线性表的第  $k$  项就在 `list[k-1]` 中。但这样做，需要将线性表项的位置减 1，才能得到含有该项的数组位置的下标。为了减少混乱及出错的机会，我们将线性表的第一项放在 `list[1]` 中，而第  $k$  项放在 `list[k]` 中。所以我们不使用——将忽略——数组元素 `list[0]`，这与段 11.1 中忽略椅子 0 几乎一样。这样做的结果是浪费了一点点的空间，但换得的是更直观的实现。

## 11.5

类 `AList` 的形式列在程序清单 11-1 中。注意类的整体结构、私有数据和构造方法。当分配数组时，我们再次使用类 `Object`，但必须将它转型为有泛型类型  $T$  的项的数组。还要注意第一个 `add` 方法及 `toArray` 方法的实现。这两个方法是我们的核心方法，因为我们将用 `add` 来建立线性表，并使用 `toArray` 来检查线性表的内容。稍后提供其他方法的实现。



**注：**类 `AList` 不是终极类，当在第 18 章讨论继承时，允许用它作为基类。下一章定义的类 `LLList` 也是一样的。

#### 程序清单 11-1 类 `AList`

```

1 import java.util.Arrays;
2 /**
3 A class that implements a list of objects by using an array.
4 Entries in a list have positions that begin with 1.
5 Duplicate entries are allowed.
6 */
7 public class AList<T> implements ListInterface<T>
8 {
9 private T[] list; // Array of list entries; ignore list[0]
10 private int numberEntries;
11 private boolean integrityOK;
12 private static final int DEFAULT_CAPACITY = 25;
13 private static final int MAX_CAPACITY = 10000;
14
15 public AList()
16 {
17 this(DEFAULT_CAPACITY); // Call next constructor
18 } // end default constructor
19
20 public AList(int initialCapacity)
21 {
22 integrityOK = false;
23
24 // Is initialCapacity too small?
25 if (initialCapacity < DEFAULT_CAPACITY)
26 initialCapacity = DEFAULT_CAPACITY;
27 else // Is initialCapacity too big?
28 checkCapacity(initialCapacity);
29
30 // The cast is safe because the new array contains null entries
31 @SuppressWarnings("unchecked")
32 T[] tempList = (T[])new Object[initialCapacity + 1];
33 list = tempList;
34 numberEntries = 0;
35 integrityOK = true;
36 } // end constructor

```

```
37 public void add(T newEntry)
38 {
39 checkIntegrity();
40 list[numberOfEntries + 1] = newEntry;
41 numberOfEntries++;
42 ensureCapacity();
43 } // end add
44
45 public void add(int givenPosition, T newEntry)
46 { < Implementation deferred >
47 } // end add
48
49 public T remove(int givenPosition)
50 { < Implementation deferred >
51 } // end remove
52
53 public void clear()
54 { < Implementation deferred >
55 } // end clear
56
57 public T replace(int givenPosition, T newEntry)
58 { < Implementation deferred >
59 } // end replace
60
61 public T getEntry(int givenPosition)
62 { < Implementation deferred >
63 } // end getEntry
64
65 public T[] toArray()
66 {
67 checkIntegrity();
68
69 // The cast is safe because the new array contains null entries
70 @SuppressWarnings("unchecked")
71 T[] result = (T[])new Object[numberOfEntries];
72 for (int index = 0; index < numberOfEntries; index++)
73 {
74 result[index] = list[index + 1];
75 } // end for
76
77 return result;
78 } // end toArray
79
80 public boolean contains(T anEntry)
81 { < Implementation deferred >
82 } // end contains
83
84 public int getLength()
85 {
86 return numberOfEntries;
87 } // end getLength
88
89 public boolean isEmpty()
90 {
91 return numberOfEntries == 0; // Or getLength() == 0
92 } // end isEmpty
93
94 // Doubles the capacity of the array list if it is full.
95 // Precondition: checkIntegrity has been called.
96 private void ensureCapacity()
97 {
98 int capacity = list.length - 1;
99 if (numberOfEntries >= capacity)
```

```

101 {
102 int newCapacity = 2 * capacity;
103 checkCapacity(newCapacity); // Is capacity too big?
104 list = Arrays.copyOf(list, newCapacity + 1);
105 } // end if
106 } // end ensureCapacity
107 < This class will define checkCapacity, checkIntegrity, and two more private
108 methods that will be discussed later. >
109 } // end AList

```

## 11.6

**核心方法。**如刚提到的，我们在实现其他方法之前，选择先实现第一个 add 方法和 toArray 方法，因为它们是类的重要部分或核心。将新项添加到线性表尾是简单的；只需紧邻在数组最后占用位置的后面添加项即可。当然，仅当数组有可用空间时才允许添加新项。

为保证至少第一次添加时有空间，构造方法创建其容量至少有 DEFAULT\_CAPACITY 的线性表。每次添加后，如果需要，add 方法则调用私有方法 ensureCapacity 来扩展数组。所以，除了检查数组容量的时间点以外，第一个 add 方法的实现非常类似于你在第 2 章的段 2.40 中看过的为类 ResizableArrayBag 所实现的方法。而且，程序清单 11-1 最后的 ensureCapacity 的定义及方法 toArray 的定义，类似于 ResizableArrayBag 中的对应方法。

### 设计决策：方法 add 应当何时调用 ensureCapacity？

之前分别在第 2 章和第 6 章给出的类 ArrayBag 和 ArrayStack 中所定义的，将项添加到基于数组的包和栈的方法，能确保在向数组中添加其他的项之前，数组有足够的容量。所以，向数组中进行添加操作时，或许必须要等待有一个更大的数组可用时。不过，当添加操作让数组变满时，在下一次调用 add 时才扩展数组。

对于线性表，我们采用相反的方法。我们选择在向数组添加另外一项后，通过调用 ensureCapacity 来修改数组的容量。因为构造方法创建的数组有足够的空间，至少能应对第一次的添加，而我们可以在添加操作装满数组后立即扩大数组。所以，数组永远为接收下一项做好了准备。不过如果没有出现再一次的添加，则数组扩展就是不必要的。

在本书目前介绍的情况下，调用 ensureCapacity 的时间点无关紧要。但是，同时执行某些任务或称并行，是当今可行及常见的技术。对于后调用 ensureCapacity，可以将其作为一个独立的线程 (thread) 让数组在后台扩展。这能让方法 add 返回，故数组扩展时客户可继续执行其他的代码。使用这种方式，可以确保当客户必须添加一项时有足够的数组容量，且不会延迟。但创建一个独立的线程不在我门讨论的范围内。

 **注：**因为 AList 的每个构造方法都创建了容量至少是 DEFAULT\_CAPACITY 的线性表，所以方法 add 可以假定，至少对线性表的第一次添加，数组有足够的空间。

## 11.7

**测试部分实现。**现在应该编写 main 方法来测试此时已经完成的代码。应该在一个类的全部实现完成之前就开始测试它。为避免程序清单 11-1 中未完成的类有语法错误，让没有完成的方法使用第 2 章段 2.16 中描述的存根。像是 getLength 和 isEmpty 方法，可以提供方法的实际定义，因为它们应该与存根一样简单。

随着你定义了更多的方法，在 main 中添加语句来测试它们。正如附录 B 中所说明的，可以将自己的方法 main 放在 AList 的定义中，供将来使用和参考。



**学习问题 4** 写方法 `displayList`, 显示线性表中的所有项。这个方法应该作为测试核心方法的驱动程序的一部分。

**在线性表的给定位置添加。**ADT 线性表有另外一个将新项添加到线性表中的方法, 将项添加在客户指定的位置。将新项添加在线性表中的任意位置, 类似于段 11.2 的示例中让学生进入到房间 A 中。虽然示例中将学生按他们姓名的字典序排列, 但请记住, 线性表的客户——不是线性表本身——来决定每个项所在的位置。所以, 如果插入位置位于线性表尾之前, 则我们必须移动数组中现有的一些项, 腾空所需的位置, 以便它能容纳新的项。如果添加在线性表尾, 则不需要这样的移动。两种情况下, 都必须要有能用来容纳新项的空间。

`add` 的下列实现使用私有方法 `makeRoom`, 来处理数据在数组内移动的细节。记住, 我们可以将项添加到线性表从 1 到其长度加 1 的位置中。根据前一章段 10.9 中给出的方法的规范说明, 如果所给位置是无效的, 则必须抛出异常。

```
public void add(int givenPosition, T newEntry)
{
 checkIntegrity();
 // Assertion: The array list has room for another entry.
 if ((givenPosition >= 1) && (givenPosition <= numberEntries + 1))
 {
 if (givenPosition <= numberEntries)
 makeRoom(givenPosition);
 list[givenPosition] = newEntry;
 numberEntries++;
 ensureCapacity(); // Ensure enough room for next add
 }
 else
 throw new IndexOutOfBoundsException(
 "Given position of add's new entry is out of bounds.");
} // end add
```

**私有方法 `makeRoom`。**下一个问题要求你实现私有方法 `makeRoom`。大多数情况下, 方法将线性表项向着数组尾的方向移动, 从最后一项开始, 如图 11-4 所示。但是, 如果 `givenPosition` 等于 `numberEntries + 1`, 那么添加发生在线性表尾, 所以不需要移动。这种情形下, `makeRoom` 什么也不做。

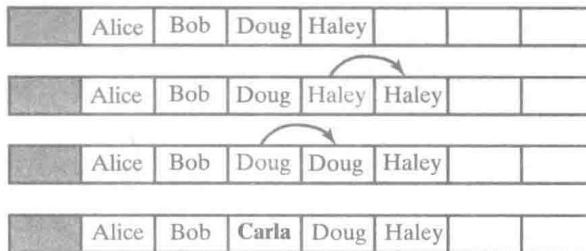


图 11-4 为将 Carla 插入为数组第三项而腾空间



**学习问题 5** 定义私有方法 `makeRoom`。包括它的前置条件, 并在测试期间使用 `assert` 语句来验证它们。要强制调用 `makeRoom` 的方法服从这些前置条件吗?

**学习问题 6** 可以实现第一个 `add` 方法, 它通过调用第二个 `add` 方法将项添加到线性表尾, 如下所示。

```
public void add(T newEntry)
{
```

11.8

11.9

```

 add(numberOfEntries + 1, newEntry);
} // end add

```

讨论这个修改后方法的优缺点。

**学习问题 7** 假定 myList 是含有 5 个项 a b c d e 的线性表。

- 执行 myList.add(5, w) 后, myList 中的内容是什么?
- 从最初的 5 个项开始, 执行 myList.add(6, w) 后 myList 中的内容是什么?
- 本学习问题的 a 和 b 中, 哪个操作需要移动数组中的项?

**学习问题 8** 如果 myList 是含 5 个项的线性表, 下列每个语句都将新项添加到线性表尾:

```

myList.add(newEntry);
myList.add(6, newEntry);

```

哪个语句需要的操作更少?

## 11.10

**方法 remove。** 在充分测试两个 add 方法和 toArray 方法后, 可以开始定义其他的方法了。删除线性表中任意位置的项类似于段 11.3 的示例中一位学生离开房间 A 时的响应。我们需要移动已有的项, 以避免数组中有空隙, 除非删除的是线性表最后一项。下列实现使用一个私有方法 removeGap, 来处理数据在数组中移动的细节。与方法 add一样, remove 负责检查给定位置的有效性。还要注意, 这个检查是如何保证线性表是不空的。

```

public T remove(int givenPosition)
{
 checkIntegrity();
 if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
 {
 // Assertion: The list is not empty
 T result = list[givenPosition]; // Get entry to be removed
 // Move subsequent entries toward entry to be removed,
 // unless it is last in list
 if (givenPosition < numberOfEntries)
 removeGap(givenPosition);
 list[numberOfEntries] = null;
 numberOfEntries--;
 return result; // Return reference to removed entry
 }
 else
 throw new IndexOutOfBoundsException(
 "Illegal position given to remove operation.");
} // end remove

```



**学习问题 9** 方法 remove 没有显式检查空线性表, 但为什么方法中给定的断言仍是正确的?

**学习问题 10** 当线性表为空时, remove 如何抛出一个异常?

## 11.11

**私有方法 removeGap。** 下列私有方法 removeGap 将线性表项在数组内移动, 如图 11-5 所示。从要被删除的项之后的一项开始, 一直到线性表尾, removeGap 将每个项移动到其相邻的更低的位置处。

```

// Shifts entries that are beyond the entry to be removed to the
// next lower position.
// Precondition: 1 <= givenPosition < numberOfEntries;
// numberOfEntries is list's length before removal;

```

```
// checkIntegrity has been called.
private void removeGap(int givenPosition)
{
 int removedIndex = givenPosition;
 for (int index = removedIndex; index < numberOfEntries; index++)
 list[index] = list[index + 1];
} // end removeGap
```

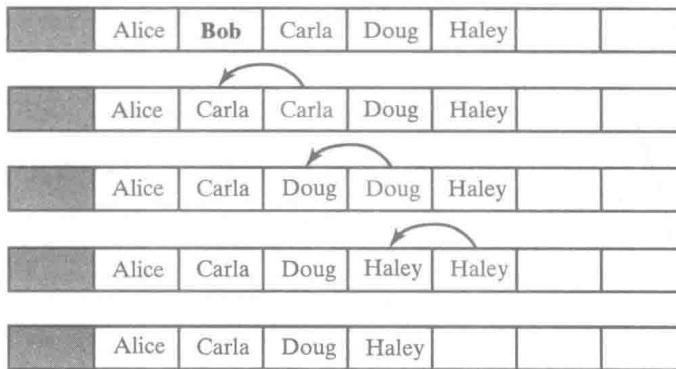


图 11-5 通过移动数组项来删除 Bob

注意到，如果删除位置位于线性表尾则不需要移动。那种情形下，线性表最后一项位于 `numberOfEntries` 位置，因为第一项是位置 1。如果 `givenPosition` 等于 `numberOfEntries`，则这种情况下 `remove` 不会调用 `removeGap`。注意到，`removeGap` 的前置条件要求 `givenPosition` 要小于 `numberOfEntries`，且 `remove` 方法强制保证这个前置条件。调试时，可以用 `removeGap` 中的 `assert` 语句来验证这个强制条件。

`removeGap` 的这个前置条件隐含着如果线性表为空时不应该调用该方法。实际上，`remove` 能保证遵守这个约定。

**学习问题 11** 图 11-5 显示 Haley 向数组开头的方向移动。实际上，指向 Haley 的引用被拷贝而不是被移动到新位置。我们应该将 Haley 的原位置赋值为 `null` 吗？

**学习问题 12** 方法 `clear` 可以简单地将数据域 `numberOfEntries` 设置为 0。虽然线性表方法能有正确的动作，就好像线性表是空的一样，但是线性表中的对象依然保留了分配的空间。给出至少两种方法，让 `clear` 能够释放这些对象。

**方法 `replace` 和 `getEntry`**。当用数组表示项时，替换线性表项及获取线性表项是两个简单的操作。可以简单地替换或获取所标识的数组位置中的对象就可以了。与前几个方法一样，`replace` 和 `getEntry` 都要负责给定位置的有效性。与 `remove` 一样，这些方法也不需要显式测试对空线性表是否有正确的动作。调试时，可以使用 `assert` 语句来验证这个断言。

**学习问题 13** 定义公有方法 `replace` 和 `getEntry`。

**方法 `contains`**。方法 `getEntry` 通过直接访问相应的数组元素来找到给定位置的项。相反，方法 `contains` 是给定了一个项，而不是项的位置，所以必须在数组中查找这个项。从下标 1 开始，方法检查每个数组项，直到或者找到所需的项，或是到达了线性表尾但没找到。下列实现中，当找到所需项时，使用局部布尔变量来中断循环：

11.12

11.13

```

public boolean contains(T anEntry)
{
 checkIntegrity();
 boolean found = false;
 int index = 1;
 while (!found && (index <= numberEntries))
 {
 if (anEntry.equals(list[index]))
 found = true;
 index++;
 } // end while
 return found;
} // end contains

```

在数组中查找给定项的这个方法称为顺序查找 (sequential search)。第 19 章进一步讨论这项技术，并提出通常情况下更快的另一个算法。

## 使用数组实现 ADT 线性表的效率

在探讨 ADT 线性表的另一种实现之前，先来评估类 AList 中几个方法的时间复杂度。

**11.14 添加到线性表尾。**从将新项添加到线性表尾的操作开始。程序清单 11-1 提供了这个操作的下列定义：

```

public void add(T newEntry)
{
 checkIntegrity();
 list[numberEntries + 1] = newEntry;
 numberEntries++;
 ensureCapacity();
} // end add

```

如果用于线性表项的数组不满，则这个方法中的每一步都是  $O(1)$  操作。运用第 4 章段 4.16 和段 4.17 中提供的知识，我们可以知道，如果数组没有扩大，则这个方法是  $O(1)$  的。即我们可以将项添加到线性表尾，而与线性表中的其他项无关。

正如我们在第 10 章看到的，变长数组是  $O(n)$  操作。所以，如果方法 ensureCapacity 遇到一个满数组，它应该需要  $O(n)$  的时间。这种情形下，添加到线性表尾的操作应该是一个  $O(n)$  操作。如果继续向线性表尾添加，则操作又应该是  $O(1)$  的。

**11.15 添加在线性表中给定的位置处。**第二个 add 方法将新项添加到线性表中由客户指定的位置。我们回忆段 11.8 中的定义：

```

public void add(int givenPosition, T newEntry)
{
 checkIntegrity();
 if ((givenPosition >= 1) && (givenPosition <= numberEntries + 1))
 {
 if (givenPosition <= numberEntries)
 makeRoom(givenPosition);
 list[givenPosition] = newEntry;
 numberEntries++;
 ensureCapacity();
 }
 else
 throw new IndexOutOfBoundsException(
 "Given position of add's new entry is out of bounds.");
} // end add

```

这个方法不同于前面的 add 方法，因为它通常会调用私有方法 makeRoom 来为新项在数

组中腾地方。唯一不调用 `makeRoom` 的时候，是当添加操作位于线性表尾的时候。回忆学习问题 5 要求定义 `makeRoom`。

在去掉 `makeRoom` 的 `assert` 语句及注释后，剩下如下的代码：

```
private void makeRoom(int givenPosition)
{
 int newIndex = givenPosition;
 int lastIndex = numberofEntries;
 for (int index = lastIndex; index >= newIndex; index--)
 list[index + 1] = list[index];
} // end makeRoom
```

当 `newPosition` 是 1 时，方法需要最多的时间，因为它必须移动线性表中的所有项。如果线性表中含有  $n$  个项，则循环体将重复  $n$  次。所以，最差情况下，方法 `makeRoom` 是  $O(n)$  的。

我们已经知道，根据是否变长数组，`ensureCapacity` 或者是  $O(n)$  的或者是  $O(1)$  的。另外，`add` 方法中的其他任务，包括将新项赋给数组元素，都是  $O(1)$  的操作。这些观察意味着，最差情况下，方法 `add` 也是  $O(n)$  的。最优情况发生在数组不需要变长且 `givenPosition` 等于 `numberofEntries + 1` 时——即当添加操作位于线性表尾时——因为不需要调用 `makeRoom`。所以，最优情况下 `add` 是  $O(1)$  的。

 **注：**添加到基于数组的线性表的开头是  $O(n)$  操作。如果底层数组不需要变大，则添加到表尾是  $O(1)$  的；否则是  $O(n)$  的。添加在其他位置的操作所需的时间与位置相关。当位置越大，添加所需的时间越少。



**学习问题 14** 线性表方法 `remove` 最优情况和最差情况的大  $O$  表示分别是什么？

**学习问题 15** 对线性表方法 `replace`，重做学习问题 14。

**学习问题 16** 对线性表方法 `getEntry`，重做学习问题 14。

**学习问题 17** 对线性表方法 `contains`，重做学习问题 14。



**注：**在第 6 章，我们使用向量——即类 `java.util.Vector` 的实例——替代数组来保存栈中的项。你可以使用向量来保存线性表中的项。因为 Java 的类 `Vector` 在其实现中使用了数组，所以 ADT 线性表基于向量的实现将以数组为基础。与 `AList` 一样，它使用变长数组来保存线性表中的项，所以线性表可以按需加大空间。



**注：**第 10 章项目 1 要求你定义实现 `ListInterface` 接口并使用向量保存线性表项的类。这样一个类中的方法应该类似于 Java 类 `Vector` 中的方法，但它们的规范说明是不同的。事实上，它们应该简单调用类 `Vector` 中的方法，所以它们的类应该是在附录 C 的段 C.3 中描述的适配器类的示例。



**注：**当使用数组或向量实现 ADT 线性表时，

- 获取给定位置的项是快的
- 将项添加到线性表尾是快的
- 添加或删除其他项中间的项，需要在数组内移动项
- 扩大数组或是向量的大小需要拷贝项

## 本章小结

- 本章的 ADT 线性表的实现使用数组来保存线性表中的项。
- 使用数组使得线性表的实现简单，但有时比 ADT 包或 ADT 栈的实现更复杂些。
- 数组提供对其任一元素的直接访问，故 `getEntry` 这样的方法有简单且高效的实现。
- 添加或删除基于数组实现的线性表项时，一般地需要其他的项在数组内移动一个位置。这个数据移动动作降低了这些操作的时间效率，特别是当线性表很长且添加或删除的位置接近线性表头时。
- 扩展数组的大小，使 `add` 方法所需的时间更长，因为做这些需要将数组内容拷贝到更大的数组中。

## 练习

1. 为类 `AList` 添加一个构造方法，能从给定的对象数组创建线性表。
2. 考虑第 10 章练习 4 描述的方法 `getPosition`。为类 `AList` 实现这个方法。
3. 考虑第 10 章练习 5 描述的方法 `remove`。如果线性表中含有 `anObject`，则方法返回真，且对象将被删除。为类 `AList` 实现这个方法。
4. 考虑第 10 章练习 6 描述的方法 `moveToEnd`。为类 `Alist` 实现这个方法。
5. 第 10 章的练习 8 要求你写客户层的语句，来替换给定线性表中的一个对象。写一个客户层的方法，完成这样一个替换。你的方法与 ADT 线性表的 `replace` 方法相比较结果如何？
6. ADT 线性表的 `replace` 方法返回被替换的对象。为类 `AList` 实现返回一个布尔值的方法 `replace`。你能不修改 `ListInterface` 而完成任务吗？
7. 假定线性表含有 `Comparable` 对象。为类 `AList` 实现下列方法：

```
/** Returns the smallest object in this list. */
public T getMin()

/** Removes and returns the smallest object in this list. */
public T removeMin()
```

8. 为 ADT 线性表实现 `equals` 方法，当一个线性表中的各项等于第二个线性表中的各项时方法返回真。特别为类 `AList` 增加这个方法。
9. 重复前一练习，但让 `equals` 方法调用私有的递归方法进行相等性检测。
10. 考虑类 `AList` 中的方法 `contains`。修改方法的定义，让它调用一个私有的递归方法，检测线性表中是否含有给定的对象。
11. 类 `AList` 有一个数组，当向线性表中添加对象时它可以增大。考虑类似的类，当从线性表中删除对象时其数组也可以变小。实现这个任务将需要两个新的私有方法。

第一个新方法检查我们是否应该减小数组的尺寸：

```
private boolean isTooBig()
```

如果线性表中项的个数小于数组大小的一半且数组大小大于 20 时，方法返回真。

第二个新方法创建一个容量为当前数组  $3/4$  的新数组，然后将线性表中的对象拷贝到新数组中：

```
private void reduceArray()
```

为我们的新类实现这两个方法。然后使用这些方法来定义方法 `remove`。

12. 考虑前一个练习中描述的两个私有方法。

- a. 方法 `isTooBig` 需要数组的大小大于 20。如果去掉这个要求会出现什么问题？
- b. 方法 `reduceArray` 不是类似 `ensureCapacity` 那样减半数组大小。如果数组大小减半而不

是减为 3/4，则会出现什么问题？

13. 对第 8 章程序清单 8-2 中所给的 `ArrayQueue` 类做什么修改，会让变长数组出现在项入队后而不是入队前？
14. 考虑类 `AList` 中的 `add` 方法。有时方法让数组变长，但有时数组不变长。数组变长会影响方法的平均时间效率吗？使用大  $O$  讨论你的答案。

## 项目

1. 写程序，充分测试类 `AList`。包括回答第 10 章练习 1 和练习 2 的方法。
2. 使用类 `AList` 的实例来保存项，定义包的类。然后写程序，充分说明这个新类。
3. 重做项目 2，但定义栈的类。
4. 重做项目 2，但定义队列的类。
5. 重做项目 2，但定义双端队列的类。
6. 重做项目 2，但定义集合的类。回忆第 1 章项目 1，集合是一个其项互异的包。
7. 考虑第 10 章项目 1 中要求你定义的类 `OurList`。
  - a. 与 `AList` 类相比，这个实现的优缺点是什么？
  - b. 将练习 1、2、3、4、6、7、8 和 9 中描述的方法添加到 `OurList` 中。
8. 使用数组，不忽略数组的第一个位置，实现接口 `ListInterface`。即将线性表的第  $i$  项保存在数组下标  $i-1$  处。
9. 使用定长数组实现 ADT 线性表，限制了线性表的大小。有些应用可以使用有有限长度的线性表。例如，飞机乘客线性表的长度，或是一场电影的持票人的线性表，都不应该超出已知的最大值。  
定义类似于 `ListInterface` 的接口 `FixedSizeListInterface`，但添加方法 `isFull`，并按需修改其他方法的规范说明，以适应定长的线性表。考虑新接口是否应该继承于 `ListInterface`。然后定义并说明实现 `FixedSizeListInterface` 的类。
10. 实现第 1 章项目 5 描述的投标集合。
11. 重做第 10 章项目 8，但使用 `AList` 的实例替代 `ArrayList`。
12. 重做第 10 章项目 10，但使用 `AList` 的实例替代 `ArrayList`。
13. 重做第 4 章项目 5，但使用 `AList` 的实例替代包的实例。
14. 修改第 10 章程序清单 10-1 中给出的 `ListInterface`，对每个方法 `add`、`remove`、`replace` 和 `getEntry`，当传给它的位置超出范围时，方法返回 `null` 或是假，而不是抛出一个异常。然后修改类 `AList`，让它实现你修改后的接口。
15. 流行的社交网络 Facebook 是由 Mark Zuckerberg 和他在哈佛大学的同班同学于 2004 年共同创建的。那时，他是学习计算机科学的大二学生。

设计并实现一个维护简单社交网络数据的应用。网络中的每个人都应该有一个简历，含有人的姓名、可选的照片、现状及朋友列表。你的应用应该允许用户加入网络、离开网络、创建简历、修改简历、查找其他人的简历及添加朋友。

16. 使用 `AList`，实现第 10 章项目 17 所设计的程序。
17. 根据 `ListInterface` 中的规范说明，使用变长数组实现 ADT 线性表，数组变长的方式如下。当数组满了，创建一个新的等大的空数组，用作原数组的扩展。当第一个扩展满了，再创建第二个扩展，如此按需扩展。图 11-6 图示了一个数组和它的扩展。

这个方法在时间和空间上比本章讨论的基于变长数组的实现的效率都高，因为项不需要复制，新数组更小些，不再需要的用于扩展的内存可以回收。

管理扩展提供了几个有意思的挑战。例如，如何记录扩展？你能用另一个类型的 ADT 来管理它们吗？你必须修改添加和删除的方法以便让项在扩展之间移动吗？

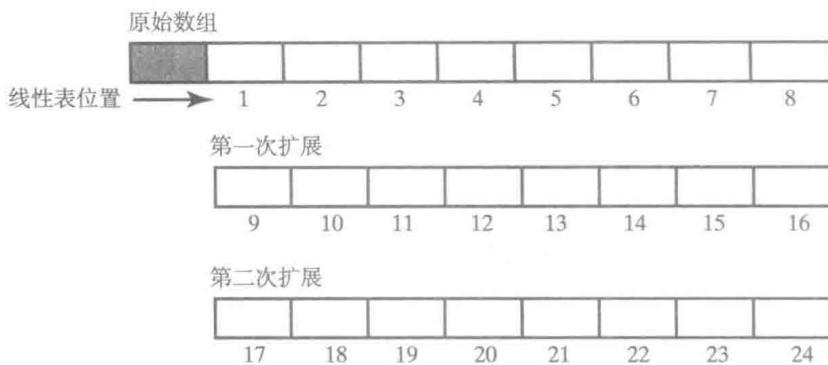


图 11-6 用于项目 17 的数组和扩展

- 18.(游戏) 实现与第 1 章项目 4 描述的 ADT 盒一样的类 Shoe。提示：要洗牌，使用牌的两个私有线性表，一个源线性表和一个洗后线性表。将所有可用的牌放到源线性表中。初始时，它含有每张牌。之后，仅有没被玩家拿着的牌才可用。使用类 `java.util.Random`，重复生成源线性表中的一个随机位置，删除那个位置的牌，然后将它放到洗后线性表的表尾。
- 写程序，充分说明类 Shoe 的操作。
- 19.(游戏) 当你玩棋牌游戏时，或当你使用共享的计算资源时，获得一轮，然后等待，直到其他的每个人都获得一轮。虽然游戏中的玩家数量保持相对静止，但共享计算服务的用户数量是有波动的。我们假定这个波动定会出现。

设计并实现一个 ADT，记录一组人的轮次。应该能添加或删除人，能查看下一次该轮到谁。实现中使用基于数组的线性表。还设计一个用来表示人的 ADT。可以保守估计这个 ADT 中所含数据的个数。第一个 ADT 将存储 ADT 人的实例。

写一个程序，充分使用——所以也是测试——你的 ADT。开始时给定一组人；为这组人指定初始的次序，次序可以是随机的也可以是用户指定的。首个加入组中的新人，应该在所有其他人都有相同轮数后获得一轮。后来的每个新人在最近入组的人获得一轮后才轮到自己。你的程序应该处理几次添加和删除，且应该说明人们得到了正确的轮次。

# 使用链式数据实现线性表

先修章节：第 3 章、第 8 章、第 10 章、第 11 章

## 目标

学习完本章后，应该能够

- 描述数据的链式组织
- 使用结点链表实现 ADT 线性表
- 讨论所给实现的优缺点
- 对比 ADT 线性表基于数组的实现和链式实现

使用数组实现 ADT 线性表，优点和缺点共存，正如你在第 11 章见到的。数组有固定的大小，且当它满时要移到一个更大的数组中。因为定长数组可能导致线性表满，所以我们的类 `AList` 使用变长数组根据线性表的需求提供空间。但是这个策略需要在扩展数组时移动数据。另外，为新项腾出空间或去掉删除后留下的空隙，数组都需要移动数据。

本章描述线性表的链式实现。与前面的链式实现一样，这种实现根据新项的需要使用内存，并在删除项后将不需要的内存返回给系统。另外，当添加或删除线性表项时它避免了数据移动。这些特征使得线性表的这种实现方式是基于数组实现方案的重要替代。

## 结点链表上的操作

我们在第 3 章使用结点链表实现了 ADT 包，在第 6 章实现了 ADT 栈。那两种情形下，都在链表的开头添加及删除结点。在第 8 章为实现 ADT 队列将结点添加在链表尾。这些操作也是线性表所必需的，我们还要能将结点添加在已有结点之间，且能从非头尾的位置删除一个结点。下面来讨论这些操作。我们使用第 3 章程序清单 3-4 中用过的同一个类 `Node`。

## 在不同的位置添加结点

为将结点添加在链表的指定位置，必须考虑下列情形：

- 情形 1：链表为空
- 情形 2：将结点添加在链表表头
- 情形 3：将结点添加在两个相邻结点之间
- 情形 4：将结点添加在链表表尾

正如后面将看到的，可以将这 4 种情形归为两类。为此，我们将检查每种情形，不过有些细节你会很熟悉。

**情形 1：将结点添加到空链表中。**虽然我们之前已将结点添加到空链表中，现在还是回忆一下必需的步骤。如果 `firstNode` 是链表的头引用，当链表为空时它的值是 `null`。图 12-1a 说明了这种状态，此外还有我们要添加到链表中的一个结点。

下列伪代码将给定的数据——由 `newEntry` 指向的——放到新结点中，然后将结点插入到空链表中：

`newNode` 指向一个新的`Node`实例  
将`newEntry` 放到 `newNode`中  
`firstNode=newNode`的地址



图 12-1 将一个结点添加到空链表中

图 12-1b 显示了这个操作的结果。在 Java 中，这些步骤如下所示：

```
Node newNode = new Node(newEntry);
firstNode = newNode;
```

注意到，在图 12-1b 中，`firstNode` 和 `newNode` 指向同一个结点。新结点插入完毕后，应该只有 `firstNode` 指向它。可以将 `newNode` 设置为 `null`，但正如你看到的，`newNode` 是方法 `add` 的局部变量。因此，在 `add` 结束运行后 `newNode` 就不存在了。

**12.2 情形 2：将结点添加在链表表头。**这种情形你应该也很熟悉了。下列伪代码描述将结点添加在链表表头所需的步骤：

`newNode` 指向一个新的`Node`实例  
将`newNode`放到`newEntry`中  
将`firstNode`的值赋给`newNode`的链接域  
让`firstNode`指向`newNode`

新结点现在是首结点。图 12-2 说明了这些步骤，下列 Java 语句将实现这些步骤：

```
Node newNode = new Node(newEntry);
newNode.setNextNode(firstNode);
firstNode = newNode;
```

为简化图，我们忽略了线性表中实际的项。这些项是结点指向的对象。



图 12-2 将结点添加在链表表头



**注：**回忆一下，如图 12-1 所展示的将结点添加到空链表中，实际上与将结点添加在链表表头是一样的。

**12.3 情形 3：将结点添加在链表中两个相邻结点之间。**因为 ADT 线性表允许在两个已有项之间进行添加，所以可以将结点添加在链表中已有的两个相邻结点之间。这个任务所必需的步骤用下列伪代码描述：

`newNode`指向新结点  
将`newEntry`放到`newNode`中

让 `nodeBefore` 指向将位于新结点之前的结点  
 将 `nodeBefore` 的链接域的值赋给 `nodeAfter`  
 将 `nodeAfter` 的值赋给 `newNode` 的链接域  
 将 `newNode` 的值赋给 `nodeBefore` 的链接域

为表明新结点应该插入链表的什么位置，我们对结点从 1 开始进行编号。我们必须找到链表中给定位置的结点，并用一个引用指向它。假定方法 `getNodeAt` 完成这个任务。因为方法返回指向一个结点的引用，且类 `Node` 是线性表类的内部类，故 `getNodeAt` 是我们不应让客户使用的实现细节。所以 `getNodeAt` 应该是一个私有方法。`getNodeAt` 的规范说明如下：

```
// Returns a reference to the node at a given position.
// Precondition: The chain is not empty;
// 1 <= givenPosition <= numberOfEntries.
private Node getNodeAt(int givenPosition)
```

稍后我们定义方法。

在此期间，仅需知道 `getNodeAt` 做什么，而不必知道它是如何做的，我们可以将它用在前一个伪代码的实现中。如果 `nodePosition` 是新结点插入后的编号，则下列 Java 语句将新结点插入链表中：

```
Node newNode = new Node(newEntry);
Node nodeBefore = getNodeAt(nodePosition - 1);
Node nodeAfter = nodeBefore.getNextNode();
newNode.setNextNode(nodeAfter);
nodeBefore.setNextNode(newNode);
```

图 12-3a 显示了执行前 3 个语句后的链表，图 12-3b 显示了结点添加后的状态。这个图中，`nodePosition` 是 3。

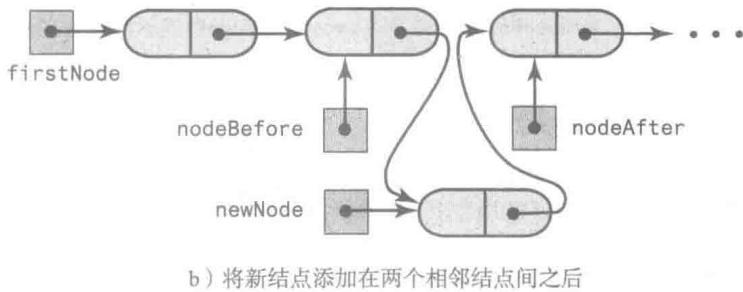
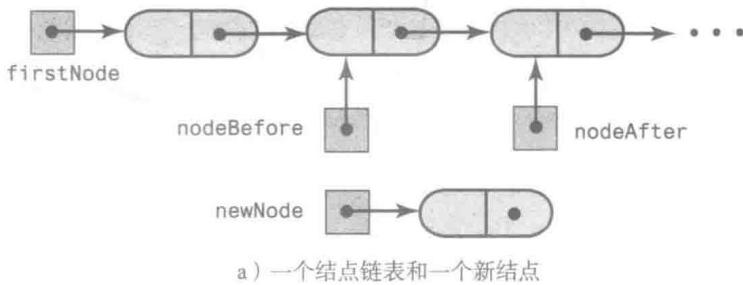


图 12-3 将结点添加在两个相邻结点之间

学习问题 1 描述方法 `getNodeAt` 为找到给定位置结点必须采取的步骤。



## 12.4

**情形 4：将结点添加在链表表尾。**要将一个结点添加在已有链表表尾，可以采取以下步骤：

newNode 指向一个新的 Node 实例

将 newEntry 放到 newNode 中

找到链表中最后一个结点

将 newNode 的地址放到这个最后结点中

即将链表的最后结点指向新结点。使用前一段描述的 getNodeAt 方法，可以在 Java 中实现这些步骤，如下所示：

```
Node newNode = new Node(newEntry);
Node lastNode = getNodeAt(numberOfEntries);
lastNode.setNextNode(newNode);
```

注意到，numberOfEntries 是链表中结点的当前个数，也是项的当前个数。图 12-4 说明了在结点链表表尾的添加过程。

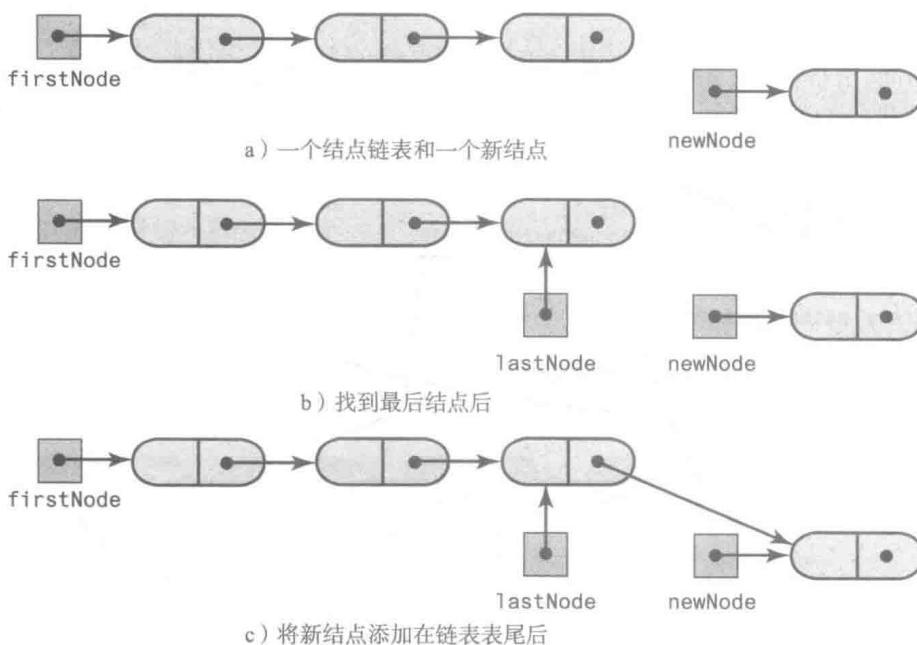


图 12-4 在链表表尾添加结点



**注：**将新结点添加到含  $n$  个结点的链表表尾，可以看作将结点添加在位置  $n+1$  处。



**学习问题 2** 段 12.3 开发的将结点添加在链表中两个相邻结点间的代码是

```
Node newNode = new Node(newEntry);
Node nodeBefore = getNodeAt(nodePosition - 1);
Node nodeAfter = nodeBefore.getNextNode();
newNode.setNextNode(nodeAfter);
nodeBefore.setNextNode(newNode);
```

能否使用这段代码替代下面我们刚刚开发的将一个结点添加到链表表尾的代码？解释你的答案。

```
Node newNode = new Node(newEntry);
Node lastNode = getNodeAt(numberOfEntries);
lastNode.setNextNode(newNode);
```

**学习问题 3** 将结点添加到空链表中，可以看作将结点添加在空链表的表尾。能使用

段 12.4 的语句来替代我们在段 12.1 开发的下列将结点添加在空链表中的代码吗？为什么？

```
Node newNode = new Node(newEntry);
firstNode = newNode;
```

## 从不同的位置删除结点

要在非空链表的指定位置删除一个结点，必须考虑两种情形：

- 情形 1：删除首结点
- 情形 2：删除首结点以外的其他结点

**情形 1：删除首结点。**这种情形你应该熟悉，与我们在 ADT 包、栈、队列、双端队列 12.5 及优先队列的链式实现中删除首结点是一样的。采取的步骤是

将首结点中的链接域的值赋给 firstNode，现在 firstNode 指向第二个结点，或者如果链表仅含一个结点，则 firstNode 的值是 null  
因为指向首结点的所有引用都不再存在，故系统自动回收首结点的内存

图 12-5 说明了这些步骤，下列 Java 语句实现了这些步骤：

```
firstNode = firstNode.getNextNode();
```

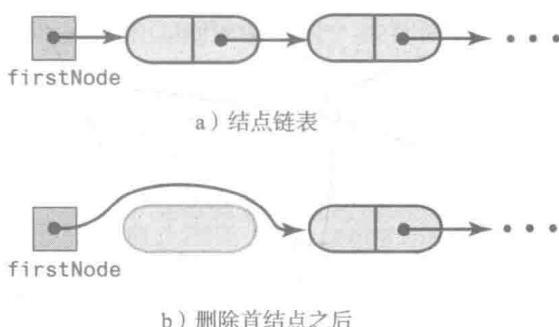


图 12-5 删除链表的首结点

**情形 2：删除首结点以外的其他结点。**第二种情形中，我们在链表的非开头位置删除一个结点。下面是采取的步骤： 12.6

让 `nodeBefore` 指向要被删除结点的前一个结点

将 `nodeBefore` 的链接域的值赋给 `nodeToRemove`，现在 `nodeToRemove` 指向要被删除的结点

将 `nodeToRemove` 的链接域的值赋给 `nodeAfter`，现在 `nodeAfter` 指向要被删除结点的后一个结点或者 null

将 `nodeAfter` 的值赋给 `nodeBefore` 的链接域（`nodeToRemove` 现在从链表中断开）

将 null 赋给 `nodeToRemove`

因为指向断开结点的所有引用都不再存在，故系统自动回收结点的内存。

下列 Java 语句实现了这些步骤，假定要删除的结点在位置 `givenPosition` 处：

```
Node nodeBefore = getNodeAt(givenPosition - 1);
Node nodeToRemove = nodeBefore.getNextNode();
Node nodeAfter = nodeToRemove.getNextNode();
nodeBefore.setNextNode(nodeAfter);
nodeToRemove = null;
```

图 12-6a 说明了前 3 条语句执行后的链表，图 12-6b 显示了结点被删除后的状态。

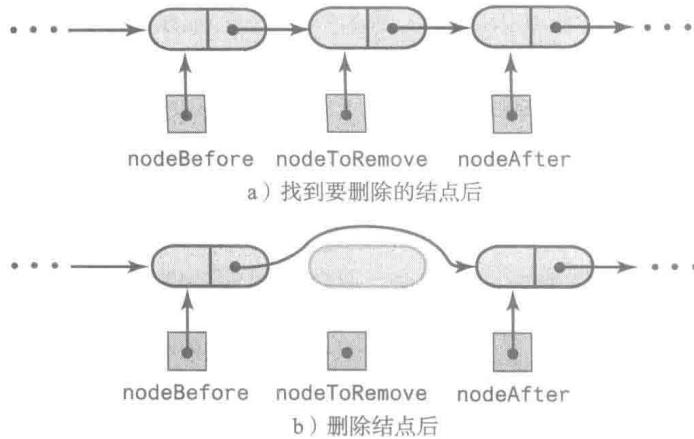


图 12-6 从链表中删除一个内部结点

## 私有方法 getNodeAt

**12.7** 前面在链表上的操作依赖于方法 `getNodeAt`, 它返回指向链表中给定位置结点的引用。回忆这个方法的规范说明:

```
// Returns a reference to the node at a given position.
// Precondition: The chain is not empty;
// 1 <= givenPosition <= numberofEntries.
private Node getNodeAt(int givenPosition)
```

要找到链表中的某个结点, 需要从链表的首结点开始, 一个结点一个结点地遍历链表。我们知道 `firstNode` 含有指向链表首结点的引用。首结点含有指向链表中第二个结点的引用, 第二个结点中含有指向第三个结点的引用, 以此类推。

可以使用一个临时变量 `currentNode`, 在从首结点遍历到要找的结点过程中, 它每次指向一个结点。初始时, 将 `currentNode` 置为 `firstNode`, 这样它指向链表的首结点。如果我们要找的是首结点, 则已经完成。否则, 通过执行

```
currentNode = currentNode.getNextNode();
```

语句移动到下一个结点。

如果我们要找的是第二个结点, 则已经完成。否则, 再次通过执行

```
currentNode = currentNode.getNextNode();
```

语句移动到下一个结点。继续这个方法, 直到找到线性表中所要找位置的结点。

`getNodeAt` 的实现如下所示。

```
private Node getNodeAt(int givenPosition)
{
 // Assertion: (firstNode != null) &&
 // (1 <= givenPosition) && (givenPosition <= numberofEntries)
 Node currentNode = firstNode;

 // Traverse the chain to locate the desired node
 // (skipped if givenPosition is 1)
 for (int counter = 1; counter < givenPosition; counter++)
 currentNode = currentNode.getNextNode();

 // Assertion: currentNode != null

 return currentNode;
} // end getNodeAt
```

在 for 循环中，如果方法的前置条件满足了，则 currentNode 永远也不会是 null。作为私有方法，getNodeAt 可以相信它的前置条件能够满足。所以如果 currentNode 是 null，则永远不会执行 currentNode.getNextNode()。在测试 getNodeAt 时可以使用 assert 语句来验证这个断言。

**STUDY** 学习问题 4 段 12.4 中将项添加到链表表尾的语句调用了方法 getNodeAt。假定你重复使用这些语句将项添加到链表表尾来创建链表。

- 这个方法的时间效率如何？
- 有没有一种更快的方法，重复地将项添加到链表表尾？解释之。

学习问题 5 getNodeAt 的前置条件如何让 currentNode 不为 null？

## 实现之初

**设计策略：应该如何高效地构造结点链表？**

假定我们有一个用来创建线性表的数据集合，即我们的数据是线性表的项。如果数据的次序就是线性表中的次序，则我们重复地将下一个项添加到线性表表尾就可以创建线性表了。

使用线性表的第一个 add 方法就可以做到。但是，如果 add 含有段 12.4 中将项添加到线性表表尾的语句，则要调用方法 getNodeAt 去查找链表中的最后一个结点。为完成这个任务，getNodeAt 必须从首结点开始进行遍历，直到找到最后一个结点。有了指向最后结点的引用，则 add 可以将新项插入链表表尾。如果方法完成时不再保留这个引用，则将另一项添加到线性表表尾时必须让 add 再次调用 getNodeAt。结果是从头开始再次遍历链表。因为我们计划将项重复地添加到线性表表尾，所以会发生很多次重复遍历。

这样的情形下，维护一个指向链表表尾的引用——及指向链表表头的引用——是有利的。这样一个指向链表表尾的引用称为尾引用，第 8 章队列的链式实现中介绍过。图 12-7 说明了两个链表：一个只带一个头引用，另一个带有头引用和尾引用。

对于线性表来说，维护头引用和尾引用有时会比队列更复杂一些，所以线性表链式实现的第一个类中没有定义尾引用。在成功完成这个简单定义后，我们将修改它，添加尾引用，从而改善其时间效率。回忆一下，首先解决简单问题常常是一个合理的策略。

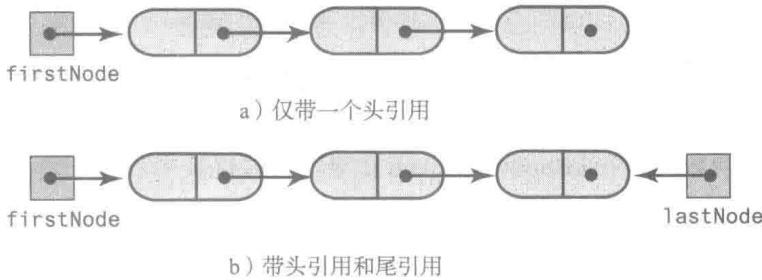


图 12-7 两个链表

## 数据域和构造方法

**12.8** 程序清单 12-1 含有实现 ADT 线性表的类 `LList`<sup>⊖</sup> 的框架。回忆一下，第 10 章定义了接口 `ListInterface`。它和实现它的类都定义了用于线性表对象的泛型 `T`。我们打算定义含有线性表项的结点链表。所以，需要用到在之前的讨论中用过的类 `Node`，故将它定义为类 `LList` 的内部类。`LList` 的类头中出现的泛型 `T` 与类 `Node` 中用到的一样。

**程序清单 12-1** 类 `LList` 的框架

```

1 /**
2 * A linked implementation of the ADT list.
3 */
4 public class LList<T> implements ListInterface<T>
5 {
6 private Node firstNode; // Reference to first node of chain
7 private int numberofEntries;
8
9 public LList()
10 {
11 initializeDataFields();
12 } // end default constructor
13
14 public void clear()
15 {
16 initializeDataFields();
17 } // end clear
18
19 < Implementation of the public methods add , remove, replace, getEntry, contains,
20 getLength, isEmpty, and toArray go here. >
21 . . .
22
23 // Initializes the class's data fields to indicate an empty list.
24 private void initializeDataFields()
25 {
26 firstNode = null;
27 numberofEntries = 0;
28 } // end initializeDataFields
29
30 // Returns a reference to the node at a given position.
31 // Precondition: The chain is not empty;
32 // 1 <= givenPosition <= numberofEntries.
33 private Node getNodeAt(int givenPosition)
34 {
35 < See Segment 12.7. >
36 } // end getNodeAt
37
38 private class Node // Private inner class
39 {
40 < See Listing 3-4 in Chapter 3. >
41 } // end Node
42 } // end LList

```

如之前所讨论的，类的这个版本将仅维护指向结点链表的头引用。数据域 `firstNode` 是这个头引用，另一个数据域 `numberofEntries` 记录当前线性表中项的个数。回忆一下，这也是链表中的结点数。默认的构造方法仅通过调用私有方法 `initializeDataFields` 来初始化这些数据域，所以初始时线性表为空，`firstNode` 是 `null`，而 `numberofEntries` 是 0。

<sup>⊖</sup> 我们将这个类命名为 `LList` 而不是 `LinkedList`，是为了避免与 `java.util` 包中的 Java 类 `LinkedList` 相混淆。本章最后你会看到 Java 的 `LinkedList`。



设计决策：LList 的构造方法和它的 clear 方法之间应该是什么关系？

LList 的构造方法和它的 clear 方法都通过调用私有方法 initializeDataFields 将类的数据域设置为相同的初值。我们可以在构造方法和 clear 方法中使用赋值语句来替代调用私有方法，与之前对第 6 章的 LinkedStack 类及第 8 章的 LinkedQueue 和 LinkedDeque 类的处理一样。虽然两种技术都遵循了合理的准则，即构造方法要对类的数据域进行显式的初始化，但 LList 还遵循了另外一个合理的准则：尽可能地重用代码。因为初始化及清空工作是简单的，所以这两种方式都可接受。本章后面我们将修改 LList，增加一个指向结点链表的尾引用。这样我们仅需修改私有方法 initializeDataFields，而不是修改构造方法和 clear 方法。

避免在 LList 的构造方法和 clear 方法中重复赋值语句的另一个办法是，让构造方法调用 clear，则 clear 的定义如下所示。

```
public final void clear() // Note the final method
{
 firstNode = null;
 numberofEntries = 0;
} // end clear
```

正如在附录 C 中所说明的，当构造方法调用像 clear 这样的另一个公有方法时，那个方法必须是终极的，以便子类不能重写它，因此也就不能改变构造方法的效果。在 clear 的类头加上 final，这是实现细节，不会反映在 ListInterface 中。回忆序言中介绍过，接口不能声明终极方法。但让构造方法调用 clear，使得构造方法初始化数据域的工作变得不那么直接。另外，我们可能想在 LList 中修改 clear 的定义——虽然在子类中不能修改——但这样的修改可能会给构造方法带来不利影响。

构造方法不应该调用 clear。初始化一个对象与清空它的数据，从概念上来讲是两个不同的动作，不能混为一谈。随后对 clear 的修改也不应该影响构造方法。第 11 章的类 AList 中，注意到构造方法与方法 clear 是无关的，因为它们执行不同的动作。

## 添加到线性表表尾

我们选择方法 add 和 toArray 作为最先实现的核心方法。12.9

先从 add 方法开始。这个方法将新项添加到线性表表尾。与段 12.4 中的语句一样，下列语句完成添加动作：

```
Node newNode = new Node(newEntry);
Node lastNode = getNodeAt(numberOfEntries);
lastNode.setNextNode(newNode);
```

假定已有私有方法 getNodeAt，它定义在段 12.7 中，则可以如下完成 add 方法：

```
public void add(T newEntry)
{
 Node newNode = new Node(newEntry);
 if (isEmpty())
 firstNode = newNode;
 else
 lastNode.setNextNode(newNode); // Make last node reference new node
 } // end if
 numberOfEntries++;
} // end add
```

这个方法先为新项创建一个新结点。如果线性表为空，则让 `firstNode` 指向它，新结点添加完成。但如果线性表不空，则必须找到线性表表尾。因为我们仅有指向首结点的引用，故必须遍历线性表，直到找到最后一个结点，并获得指向它的引用。调用私有方法 `getNodeAt` 来完成这个任务。因为数据域 `numberOfEntries` 含有线性表的大小，且我们使用从 1 开始的其在表中的位置来标识线性表项，故最后的结点在 `numberOfEntries` 位置。我们必须将这个值传给 `getNodeAt`。一旦 `getNodeAt` 返回指向最后结点的引用，我们就能将最后结点的链接设置为指向新结点的引用。

注意到，必须定义方法 `isEmpty`，因为 `add` 调用它，所以将它添加到我们最初定义的核心方法组内。

## 在线性表的给定位置添加

**12.10** 第二个 `add` 方法将新项添加在线性表内的指定位置。在检查给定位置的合理性后，创建由 `newNode` 指向的新结点。然后必须考虑两种情形：

- 情形 1：将新结点添加在链表表头
- 情形 2：将新结点添加在除链表表头外的位置

段 12.2 给出了如下语句来实现在链表表头的添加：

```
Node newNode = new Node(newEntry);
newNode.setNextNode(firstNode);
firstNode = newNode;
```

回忆一下，在链表为空及不为空的情况下，这些语句都是适用的。段 12.3 中所示的语句可以完成在任何地方的添加：

```
Node newNode = new Node(newEntry);
Node nodeBefore = getNodeAt(givenPosition - 1);
Node nodeAfter = nodeBefore.getNextNode();
newNode.setNextNode(nodeAfter);
nodeBefore.setNextNode(newNode);
```

**12.11** Java 方法。基于前面的代码段给出 `add` 方法的下列实现。先检查插入位置 `givenPosition` 的有效性。如果它在合理范围内，则创建新结点。然后根据其预期位置 `givenPosition` 将新结点插入链表中：插入或者在链表的表头，或者在链表中的其他位置。

```
public void add(int givenPosition, T newEntry)
{
 if ((givenPosition >= 1) && (givenPosition <= numberOfEntries + 1))
 {
 Node newNode = new Node(newEntry);
 if (givenPosition == 1) // Case 1
 {
 newNode.setNextNode(firstNode);
 firstNode = newNode;
 }
 else // Case 2: List is not empty and givenPosition > 1
 {
 Node nodeBefore = getNodeAt(givenPosition - 1);
 Node nodeAfter = nodeBefore.getNextNode();
 newNode.setNextNode(nodeAfter);
 nodeBefore.setNextNode(newNode);
 } // end if
 numberOfEntries++;
 }
}
```

```

 throw new IndexOutOfBoundsException(
 "Illegal position given to add operation.");
 } // end add
}

```



### 学习问题 6 考虑前面 add 方法中的第一个 else 子句。

- 能给这个子句添加什么 assert 语句？
- 用什么参数调用 getNodeAt 能替代赋给 nodeAfter 的值？
- 应该进行提出的修改吗？

**学习问题 7** 在前面的 add 方法中，第二个 if 语句测试 givenPosition 的值。它测试的布尔表达式应该是 isEmpty() || (givenPosition == 1) 吗？解释之。

**学习问题 8** 段 12.9 和段 12.11 给出的 add 方法是如何遵守 getNodeAt 的前置条件的？

**学习问题 9** 修改段 12.9 中给出的 add 方法的定义，让它调用段 12.11 中给出的 add 方法。

## 方法 isEmpty 和 toArray

**方法 isEmpty 和断言。**方法 isEmpty 的实现可以简单测试线性表的长度是不是 0，与在第 11 章看到的基于数组的实现中所做的一样。但是，这里我们使用另一个规范。当线性表为空时，firstNode 引用是 null。如果实现是正确的，那么这两个规范都很好，但开发过程中，当类的某些地方可能含有逻辑上的错误时会发生什么情况呢？可以使用临时的 assert 语句采用第二个规范来帮助我们捕获错误，与 isEmpty 的前一个版本中用到的一样：

```

public boolean isEmpty()
{
 boolean result;
 if (numberOfEntries == 0) // Or getLength() == 0
 {
 // Assertion: firstNode == null
 result = true;
 }
 else
 {
 // Assertion: firstNode != null
 result = false;
 } // end if
 return result;
} // end isEmpty

```

**示例。**我们来看一个示例，isEmpty 方法前面的实现版本如何帮助我们找到逻辑上的错误。考虑段 12.9 给出的第一个 add 方法的定义。如果我们担心忘记增加数据域 numberOfEntries 的值，那么可能会在方法的第一句就写 numberOfEntries++ 语句，而不是放在最后才写。这个修改可能导致一个错误。如果方法调用时线性表为空，则 numberOfEntries 会被赋值为 1，且将会调用 isEmpty。假定我们已启用断言，因为 firstNode 应该是 null，故 isEmpty 中的第二个断言应该产生一个像下面这样的错误信息：

```

Exception in thread "main" java.lang.AssertionError
at LList.isEmpty(LList.java:175)
at LList.add(LList.java:23)
at Driver.main(driver.java:15);

```

12.12

12.13

这条信息指出，方法 add 调用了 isEmpty，而后者产生断言错误。可以在 isEmpty 中添加 assert 语句来说明这条信息。例如，如果第二个 assert 语句如下：

```
assert firstNode != null : "numberOfEntries is not 0 but firstNode is null";
```

则前面这个错误信息的开头将如下所示：

```
Exception in thread "main" java.lang.AssertionError:
numberOfEntries is not 0 but firstNode is null
```

如果执行程序时没有启用断言，则 isEmpty 只简单测试 numberOfEntries。因为 numberOfEntries 不会是 0，isEmpty 会返回假，故将会执行 add 的 else 子句。当 add 调用 getNodeAt(1) 时，会返回 null——因为 firstNode 是 null——并赋给 lastNode。

结果是，lastNode.setNextNode(newNode) 将导致一个异常，并得到如下这样的错误信息：

```
Exception in thread "main" java.lang.NullPointerException
at LList$Node.access$102(LList.java:212)
at LList.add(LList.java:28)
at Driver.main(driver.java:15);
```

这条信息不如前一条清晰，所以还需要花更多的努力去找出问题之所在。



#### 学习问题 10 假定 isEmpty 的方法体仅含有下面这条语句：

```
return (numberOfEntries == 0) && (firstNode == null);
```

如果 add 方法中有前一段所描述的错误，当调用 add 且线性表为空时会发生什么？假定启用了断言。

12.14

**方法 toArray。**实现了 toArray 方法，就能在完成 LList 其余部分之前测试前面写的方法了。这个方法必须遍历链表，并将每个结点中的数据拷贝到数组的一个元素中。所以它需要一个局部变量指向链表中的每个结点。例如，currentNode 可以指向我们想拷贝的数据所在的结点。这个数据是 currentNode.getData()。

初始时想让 currentNode 指向链表中的首结点，所以将它设置为 firstNode。要让 currentNode 指向下一个结点，可以执行语句

```
currentNode = currentNode.getNextNode();
```

所以，可以写一个循环来进行迭代，直到 currentNode 变为 null 时为止。

将这个思想用于下面给出的 toArray 方法中：

```
public T[] toArray()
{
 // The cast is safe because the new array contains null entries
 @SuppressWarnings("unchecked")
 T[] result = (T[])new Object[numberOfEntries];
 int index = 0;
 Node currentNode = firstNode;
 while ((index < numberOfEntries) && (currentNode != null))
 {
 result[index] = currentNode.getData();
 currentNode = currentNode.getNextNode();
 index++;
 } // end while
 return result;
} // end toArray
```



**学习问题 11** 前面所实现的 `toArray` 方法中, `while` 语句测试 `index` 和 `currentNode` 的值。你能用下面的语句替换 `while` 语句吗?

- `while (index < numberEntries)`
- `while (currentNode != null)`

解释你的答案。

**学习问题 12** 将前面给出的 `toArray` 方法中循环所做的工作与下列循环所做的工作进行比较:

```
int index = 0;
Node currentNode = firstNode;
while ((index < numberEntries) && (currentNode != null))
{
 currentNode = getNodeAt(index + 1);
 result[index] = currentNode.getData();
 index++;
} // end while
```

## 测试核心方法

之前, 我们知道 `add` 方法是类的基础, 所以它们是我们要先实现并测试的核心方法部分。方法 `toArray` 能让我们查看 `add` 是否能正确工作, 所以它也在核心组内。构造方法也很基础, 方法 `initializeDataFields` 也一样, 因为构造方法要调用它。类似地, 因为 `add` 调用 `isEmpty` 和 `getNodeAt`, 故它们也属于我们要先实现并测试的核心方法。虽然 `clear` 不是核心方法, 但我们已经定义了它, 所以我们也要测试它。最后, 我们定义方法 `getLength` 用来检查 `add` 方法是否能正确维护数据域 `numberEntries`。虽然到目前为止它还是一个真正的基础方法, 但它的定义简单, 且与在第 11 章看到的基于数组的实现是相同的。

12.15

现在, 我们已经实现了这些核心方法, 可以测试它们了。但因为 `LLList` 实现了 `ListInterface`, 所以必须先写出这个接口中其他方法的存根。假定我们已经完成了这些简单任务。

我们选择第一个进行测试的方法是在线性表尾进行添加的 `add` 方法。程序清单 12-2 含有一个 `main` 方法, 是用来测试这个方法的。为了能让我们明白实现是否正确, 注意输出是如何描述的。方法 `displayList` 与第 11 章为了测试类 `AList` 的部分实现时而写的方法是一样的。在第 11 章的学习问题 4 的答案中能看到这个方法。回忆一下, 这个方法调用 `toArray`, 进而测试它。

### 程序清单 12-2 测试 ADT 线性表的部分实现的 `main` 方法

```
1 public static void main(String[] args)
2 {
3 System.out.println("Create an empty list.");
4 ListInterface<String> myList = new LLList<>();
5 System.out.println("List should be empty; isEmpty returns " +
6 myList.isEmpty() + ".");
7 System.out.println("\nTesting add to end:");
8 myList.add("15");
9 myList.add("25");
10 myList.add("35");
11 myList.add("45");
12 System.out.println("List should contain 15 25 35 45.");
```

```
13 displayList(myList);
14 System.out.println("List should not be empty; isEmpty() returns " +
15 myList.isEmpty() + ".");
16 System.out.println("\nTesting clear():");
17 myList.clear();
18 System.out.println("List should be empty; isEmpty returns " +
19 myList.isEmpty() + ".");
20 } // end main
```

输出

```
Create an empty list.
List should be empty; isEmpty returns true.

Testing add to end:
List should contain 15 25 35 45.
List contains 4 entries, as follows:
15 25 35 45
List should not be empty; isEmpty() returns false.

Testing clear():
List should be empty; isEmpty returns true.
```



**学习问题 13** 考虑第 11 章学习问题 4 的答案中给出的方法 `displayList`。当线性表是下列类的实例时，这个方法的时间效率是多少？

- a. 第 11 章给出的 AList。
  - b. 本章给出的 LList。

继续实现

为完成类 `LList`, 现在来定义方法 `remove`、`replace`、`getEntry` 和 `contains`。

12.16

方法 remove。要从线性表中删除第一项，执行下列语句：

```
firstNode = firstNode.getNextNode();
```

要删除第一项之后的项，执行下列语句

```
Node nodeBefore = getNodeAt(givenPosition - 1);
Node nodeToRemove = nodeBefore.getNextNode();
Node nodeAfter = nodeToRemove.getNextNode();
nodeBefore.setNextNode(nodeAfter);
nodeToRemove = null; // Recycle node memory
```

回忆一下，`remove`方法返回从线性表中删除的项。虽然含有项的结点被回收了，但只要客户保存指向它的引用，项本身就不被回收。

```

 Node nodeBefore = getNodeAt(givenPosition - 1);
 Node nodeToRemove = nodeBefore.getNextNode();
 result = nodeToRemove.getData(); // Save entry to be removed
 Node nodeAfter = nodeToRemove.getNextNode();
 nodeBefore.setNextNode(nodeAfter); // Remove entry
 } // end if
 numberofEntries--; // Update count
 return result; // Return removed entry
}
else
 throw new IndexOutOfBoundsException(
 "Illegal position given to remove operation.");
} // end remove

```

注意到，我们用到了原来为 `add` 方法而写的私有方法 `getNodeAt`，找到要删除结点之前的结点。仅当删除非首结点中的项时才调用这个方法。所以它的参数 `givenPosition-1` 永远大于 0，正如它的前置条件所要求的。

还要注意到，断开结点后我们没有显式地将 `nodeToRemove` 设置为 `null`。这个变量是 `remove` 方法的局部变量，所以方法运行结束后它就不存在了。虽然能将 `nodeToRemove` 设置为 `null`，但这样做没有必要。

#### 学习问题 14 前一个方法中的断言为什么是真的？



STUDY

方法 `replace`。替换线性表中的项，需要用其他的数据替代结点中的数据部分。可以使用私有方法 `getNodeAt` 找到结点，然后简单地替换它的数据部分。调用 `getNodeAt` 之前，要检查线性表不为空，且给定的位置值是有效的。方法实现如下所示。

```

public T replace(int givenPosition, T newEntry)
{
 if ((givenPosition >= 1) && (givenPosition <= numberofEntries))
 {
 // Assertion: !isEmpty()
 Node desiredNode = getNodeAt(givenPosition);
 T originalEntry = desiredNode.getData();
 desiredNode.setData(newEntry);
 return originalEntry;
 }
 else
 throw new IndexOutOfBoundsException(
 "Illegal position given to replace operation.");
} // end replace

```



注：方法 `replace` 替换结点中的数据，而不是结点本身。



学习问题 15 分别使用前面这个 `replace` 方法以及前一章段 11.12 中给出的基于数

组的版本，来替换线性表中的项，比较它们的时间需求。

#### 方法 `getEntry`。获取线性表项很简单：

```

public T getEntry(int givenPosition)
{
 if ((givenPosition >= 1) && (givenPosition <= numberofEntries))
 {
 // Assertion: !isEmpty()
 return getNodeAt(givenPosition).getData();
 }
}

```

12.18

```

 }
 else
 throw new IndexOutOfBoundsException(
 "Illegal position given to getEntry operation.");
} // end getEntry

```

方法 `getNodeAt` 返回指向所需结点的引用，故

```
getNodeAt(givenPosition).getData()
```

是结点的数据部分。

虽然 `getEntry` 和 `replace` 的实现很容易写，但每个方法都比使用数组表示线性表时做了更多的事情。这里，`getNodeAt` 从链表的首结点开始，从一个结点移到另一个结点，直到到达所需的结点。而在基于数组的实现中，`replace` 和 `getEntry` 可以直接指向所需的数组项，而不会涉及数组中的其他项。



**学习问题 16** 考虑第 10 章程序清单 10-2 中给出的方法 `displayList`。当线性表是下列类的实例时，这个方法的时间效率如何？

- a. 第 11 章给出的 `AList`。
- b. 本章给出的 `LList`。

12.19

方法 `contains`。用于线性表的方法 `contains`，可以与第 3 章段 3.17 中为包给出的方法的定义一样。但是，这里的内部类 `Node` 有设置和获取方法，所以 `contains` 如下所示：

```

public boolean contains(T anEntry)
{
 boolean found = false;
 Node currentNode = firstNode;
 while (!found && (currentNode != null))
 {
 if (anEntry.equals(currentNode.getData()))
 found = true;
 else
 currentNode = currentNode.getNextNode();
 } // end while
 return found;
} // end contains

```

因为 ADT 包有一个删除给定项的 `remove` 方法，故它和 `contains` 一样要做同样的查找。因此，我们修改第 3 章 `contains` 的定义，以便 `contains` 和 `remove` 都通过调用私有方法来执行查找。但是 ADT 线性表的版本是按位置而不是按项的值来删除项的。所以 `contains` 方法所做的查找仅由 `contains` 来执行。



### 注：测试类 `LList`

现在类 `LList` 已经完成，在继续进行下去之前应该进行充分的测试。将这个测试留作练习。过后你要使用自己写的测试程序，来测试我们就要实现的 `LList` 的改进版本。

## 完善实现

目前，含有线性表项的结点链表仅有一个头引用。当开始写类 `LList` 时，我们注意到，

将新项添加在链表表尾的第一个 add 方法，必须调用私有方法 `getNodeAt` 来找到链表的最后一个结点。为此，`getNodeAt` 必须从表头开始遍历链表。除链表的头引用外，我们还可以维护一个指向链表表尾的引用，从而改善这个 add 方法的时间效率，如之前图 12-7b 及这里重画的图 12-8 所示。使用这个办法，避免了每次调用 add 时对整个链表的遍历。

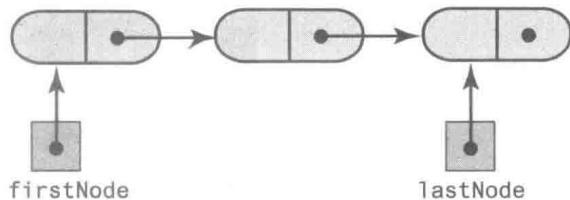


图 12-8 具有头引用和尾引用的结点链表

 **学习问题 17** 分析本章给出的类 `LList` 的实现。如果头引用和尾引用都使用，哪些方法需要新的定义？

## 尾引用

与头引用一样，尾引用是类的私有数据域。故改进后的类的私有数据域是

```
private Node firstNode; // Head reference to first node
private Node lastNode; // Tail reference to last node
private int numberOfEntries; // Number of entries in list
```

通过仔细分析本章之前描述的类 `LList`，你应该发现，两个 add 方法和 remove 方法及 `initializeDataFields` 方法，都涉及头引用和尾引用，所以都需要修改。虽然私有方法 `getNodeAt` 可以与在 `LList` 中的定义相同，不过我们修改它，以便当需要访问最后一个结点时避免遍历链表。我们还应该修改方法 `isEmpty` 中的断言。原实现中的其他部分，包括构造方法都保持不变。我们来分析这些修改。为与原来的类相区别，将改版后的类称为 `LListWithTail`。

方法 `initializeDataFields`。我们从 `initializeDataFields` 方法开始，因为构造方法调用了它。它必须初始化头引用和尾引用，及域 `numberOfEntries`:

```
private void initializeDataFields()
{
 firstNode = null;
 lastNode = null;
 numberOfEntries = 0;
} // end initializeDataFields
```

此处，及其他修改代码中，将与原实现不同的地方都标注了出来。

在线性表表尾添加。将项添加在线性表的表尾所需的步骤，依线性表是否为空而定。将项添加到空表的尾部后，头引用和尾引用都必须指向新的唯一的结点。所以，创建由 `newNode` 指向的新结点后，add 方法将执行

```
firstNode = newNode;
lastNode = newNode;
```

添加在非空线性表的表尾不再需要查找最后项的遍历：尾引用 `lastNode` 可以提供这个信息。添加完成后，必须修改尾引用，让其指向新的最后项。下列语句执行这些步骤，如图 12-9 所示：

```
lastNode.setNextNode(newNode);
lastNode = newNode;
```

12.20

12.21

12.22

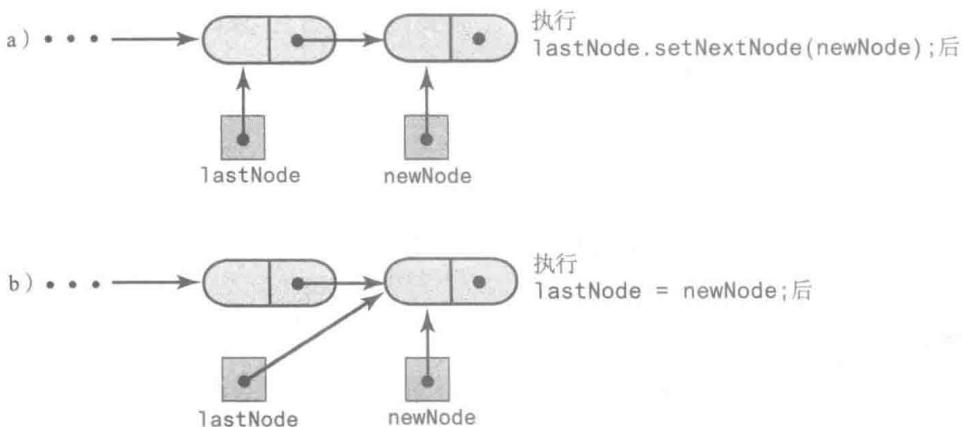


图 12-9 将结点添加到有尾引用的非空链表的表尾

第一个 add 方法的如下版本反映了前面说明的思想：

```
public void add(T newEntry)
{
 Node newNode = new Node(newEntry);
 if (isEmpty())
 firstNode = newNode;
 else
 lastNode.setNextNode(newNode);
 lastNode = newNode;
 numberOfEntries++;
} // end add
```

注意到，方法不再像段 12.9 中那样，通过调用 getNodeAt 来得到 lastNode。

12.23

**添加到链表中给定的位置。**根据位置向线性表中添加项，仅当向空线性表添加，或添加到非空线性表的表尾时才影响到尾引用。其他情形都不影响尾引用，所以处理流程与在段 12.11 中没有尾引用时所做的一样。

所以，根据位置添加项的方法的实现如下：

```
public void add(int givenPosition, T newEntry)
{
 if ((givenPosition >= 1) && (givenPosition <= numberOfEntries + 1))
 {
 Node newNode = new Node(newEntry);
 if (isEmpty())
 {
 firstNode = newNode;
 lastNode = newNode;
 }
 else if (givenPosition == 1)
 {
 newNode.setNextNode(firstNode);
 firstNode = newNode;
 }
 else if (givenPosition == numberOfEntries + 1)
 {
 lastNode.setNextNode(newNode);
 lastNode = newNode;
 }
 else
 {
 Node nodeBefore = getNodeAt(givenPosition - 1);
 Node nodeAfter = nodeBefore.getNextNode();
```

```

 newNode.setNextNode(nodeAfter);
 nodeBefore.setNextNode(newNode);
 } // end if
 numberOfEntries++;
}
else
 throw new IndexOutOfBoundsException(
 "Illegal position given to add operation.");
} // end add

```



**注：**添加到链表表尾的操作，当维护一个尾引用时要做的事情更少，因为避免了遍历链表。



**学习问题 18** 当在段 12.10 和段 12.11 中为类 `LList` 定义 `add` 方法时，将一个项添加到线性表表头的代码，即使线性表是空的，也是适用的。而 `LListWithTail` 类的 `add` 方法中，空线性表为什么是特殊情形？

从线性表中删除一项。在两种情形下项的删除可能影响到尾引用：

12.24

- 情形 1：如果线性表只含有一个项而且我们要删除它，得到空表，那么我们必须将头引用和尾引用都设置为 `null`。
- 情形 2：如果线性表含有多个项而我们删除最后一项，那么必须修改尾引用，让其指向新的最后一项。

图 12-10 说明了这两种情形。

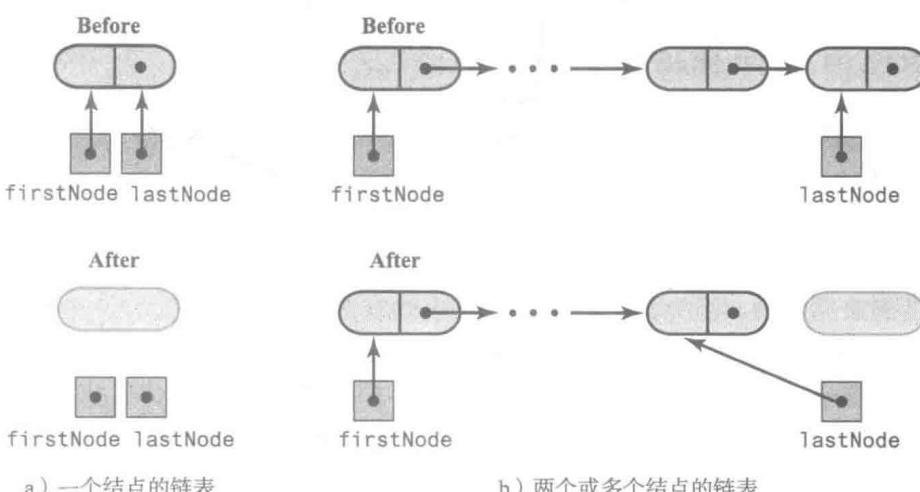


图 12-10 从有头引用和尾引用，且含有一个结点或多个结点的链表中删除最后一项之前和之后

实现删除操作时，下列方法考虑了前述的两种情形：

```

public T remove(int givenPosition)
{
 T result = null; // Return value
 if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
 {
 // Assertion: !isEmpty()
 if (givenPosition == 1) // Case 1: Remove first entry
 {
 result = firstNode.getData(); // Save entry to be removed
 firstNode = firstNode.getNext();
 firstNode.setPrevious(null);
 if (firstNode == null)
 lastNode = null;
 else
 firstNode.setPrevious(lastNode);
 }
 else
 {
 Node previous = firstNode;
 for (int i = 1; i < givenPosition - 1; i++)
 previous = previous.getNext();
 result = previous.getNext().getData();
 previous.setNext(previous.getNext().getNext());
 if (previous.getNext() == null)
 lastNode = previous;
 else
 previous.setNext(next.getNext());
 }
 }
}

```

```

 firstNode = firstNode.getNextNode();
 if (numberOfEntries == 1)
 lastNode = null; // Solitary entry was removed
 }
 else // Case 2: Not first entry
 {
 Node nodeBefore = getNodeAt(givenPosition - 1);
 Node nodeToRemove = nodeBefore.getNextNode();
 Node nodeAfter = nodeToRemove.getNextNode();
 nodeBefore.setNextNode(nodeAfter);
 result = nodeToRemove.getData(); // Save entry to be removed
 if (givenPosition == numberOfEntries)
 lastNode = nodeBefore; // Last node was removed
 } // end if
 numberOfEntries--;
}
else
 throw new IndexOutOfBoundsException(
 "Illegal position given to remove operation.");
return result; // Return removed entry
} // end remove

```



**注：**从链表中删除最后结点还需要遍历，为的是找到倒数第二个结点，不管有没有尾引用都是如此。



**学习问题 19** 鉴于尾引用的存在，在段 12.12 给出的方法 `isEmpty` 中，应该对断言进行哪些修改？

## 使用链表实现 ADT 线性表的效率

我们来考虑类 `LList` 和 `LListWithTail` 中一些方法的时间复杂度。其中有几个方法调用了段 12.7 中给出的私有方法 `getNodeAt`。在查找链表中第  $i$  个结点时，`getNodeAt` 中的循环执行了  $i-1$  次。所以一般来讲 `getNodeAt` 是  $O(n)$  的，但当跳过循环时是  $O(1)$  的。我们将这个事实在对公有方法的分析中。

12.25

**在线性表表尾添加。**因为类 `LList` 中的链表没有尾引用，所以段 12.9 中描述的 `LList` 的 `add` 方法必须遍历整个结点链表来找到最后一个结点，才能将项插入在线性表表尾。方法调用 `getNodeAt` 来查找这个结点。因为这种情形下 `getNodeAt` 是  $O(n)$  的，所以这个 `add` 方法也是  $O(n)$  的。

而另一方面，类 `LListWithTail` 为结点链表维护了尾引用。所以段 12.22 中给出的它的 `add` 方法，不去调用 `getNodeAt`，所以是  $O(1)$  的。

12.26

**在线性表的给定位置添加。**段 12.11 给出的类 `LList` 的 `add` 方法，能在  $O(1)$  时间内将项添加到线性表的开头。在线性表的任意位置的添加，取决于添加的位置。随着位置数的增大，添加所需的时间也增多。换句话说，当在线性表表头之后添加时，`add` 是  $O(n)$  的，因为这种情形下 `add` 调用了 `getNodeAt`，而 `getNodeAt` 是  $O(n)$  的。

段 12.23 中给出的类 `LListWithTail` 的 `add` 方法，可以在  $O(1)$  时间内将项添加在线性表的表头或表尾。注意，这些情况下都不调用 `getNodeAt`。对于其他的添加，`getNodeAt` 需要  $O(n)$  时间。



**学习问题 20** 段 12.14 给出的 `toArray` 方法的大  $O$  表示是多少?

**学习问题 21** 段 12.16 给出的 `remove` 方法的大  $O$  表示是多少?

**学习问题 22** 段 12.17 给出的 `replace` 方法的大  $O$  表示是多少?

**学习问题 23** 段 12.18 给出的 `getEntry` 方法的大  $O$  表示是多少?

**学习问题 24** 段 12.19 给出的 `contains` 方法的大  $O$  表示是多少?

**学习问题 25** 鉴于尾引用的存在, 为改善方法 `replace`、`getEntry` 和 `contains` 的时间复杂度, 要对段 12.7 中给出的方法 `getNodeAt` 进行哪些修改?

使用大  $O$  表示, 图 12-11 概括了使用数组和使用结点链表实现 ADT 线性表操作的时间复杂度。对某些操作, 给出了两个或三个复杂度: 第一个表示在线性表表头进行操作所需的时间, 第二个是在线性表其他位置进行操作所需的时间, 如果有第三个, 是在线性表表尾进行操作对应的时间。

12.27

| 操作                                            | AList              | LList        | LListWithTail      |
|-----------------------------------------------|--------------------|--------------|--------------------|
| <code>add(newEntry)</code>                    | $O(1)$             | $O(n)$       | $O(1)$             |
| <code>add(givenPosition, newEntry)</code>     | $O(n); O(n); O(1)$ | $O(1); O(n)$ | $O(1); O(n); O(1)$ |
| <code>toArray()</code>                        | $O(n)$             | $O(n)$       | $O(n)$             |
| <code>remove(givenPosition)</code>            | $O(n); O(n); O(1)$ | $O(1); O(n)$ | $O(1); O(n)$       |
| <code>replace(givenPosition, newEntry)</code> | $O(1)$             | $O(1); O(n)$ | $O(1); O(n); O(1)$ |
| <code>getEntry(givenPosition)</code>          | $O(1)$             | $O(1); O(n)$ | $O(1); O(n); O(1)$ |
| <code>contains(anEntry)</code>                | $O(n)$             | $O(n)$       | $O(n)$             |
| <code>clear(), getLength(), isEmpty()</code>  | $O(1)$             | $O(1)$       | $O(1)$             |

图 12-11 三种实现方式下, 以大  $O$  表示的 ADT 线性表操作的时间效率

例如, 基于数组实现的第一个 `add` 方法是  $O(1)$  的, 第二个 `add` 方法是  $O(n)$  的, 除非它在线性表尾进行添加, 而那种情形下它是  $O(1)$  的。方法 `toArray` 总是  $O(n)$  的, 而 `getEntry` 也总是  $O(1)$  的。

对于仅维护结点链表的头引用的链式实现 `LList`, 第一个 `add` 方法和 `toArray` 方法都是  $O(n)$  的。第二个 `add` 方法是  $O(n)$  的, 除非它在线性表表头进行添加, 而那种情形下它是  $O(1)$  的。

对于维护结点链表的头引用和尾引用的链式实现 `LListWithTail`, 第一个 `add` 方法是  $O(1)$  的, 而 `toArray` 方法是  $O(n)$  的。第二个 `add` 方法是  $O(n)$  的, 除非它在线性表表头或表尾进行添加, 而那两种情形下它是  $O(1)$  的。

正如你看到的, 有些操作在每种实现下都有相同的时间复杂度。但是, 添加到线性表尾的操作、替换项的操作或是获取项的操作, 当使用数组表示线性表时, 要比使用结点链表表示花费更少的时间。如果你的应用频繁用到了这些操作, 则基于数组的实现更具有吸引力。

在给定位置添加或删除项的操作, 所需的时间依赖于这个位置, 而不管是哪种实现方式。如果你的应用主要是在线性表表头或接近线性表表头的位置添加或删除项, 则使用链式实现。如果这些操作大多数是在线性表表尾或接近线性表表尾, 则使用基于数组的实现。应用中使用最多的操作将影响对 ADT 实现的选择。



### 设计策略：ADT 应该使用哪种实现方式？

即使是对 ADT 底层数据结构的微小改变，都可能增加或减少 ADT 操作的时间效率。为 ADT 选择实现方式时，应该考虑应用中所需要的操作。如果频繁用到某一种操作，则要让它的实现高效。相反，如果很少使用一种操作，则可以使用对这个操作的实现并不高效的类。

与你在前几章见过的链式实现一样，类 `LList` 和 `LListWithTail` 能使它们的实例按需增大。可以在链表中添加任意多的结点——即在线性表中添加任意多的项——只要计算机内存允许。虽然基于数组的实现中，变长数组也能带给你同样的好处，但它相伴的是将数据从一个数组拷贝到另一个数组的开销。在链式实现中不需要这样的拷贝。

另外，链表能让你在不移动线性表中现有项的情况下添加和删除结点。对于数组，添加项和删除项通常需要其他项在数组内移动。但是，你必须从表头开始遍历链表来确定添加或删除的位置。

获取链表中已有的项，需要类似的遍历，来找到所需的项。当使用数组而不是链表时，可以直接按位置访问任意元素，而不需要查找数组。但是，像 `contains` 这样的没有项的位置的方法，还是必须执行查找，而不管线性表是用数组还是用链表来表示的。

最后，正如你之前见过的，与数组相比，链表保存了额外的引用。对于线性表中的每个项，与数组相比，链表中保存了两个引用。这个额外的内存需求抵消掉一些链表按照每个线性表项的需求使用内存的这个事实优势，而数组常常大于所需的存储，所以会浪费内存。

ADT 的任何实现都有其优点和缺点。应该选择最适用于具体应用的实现方案。

## Java 类库：类 `LinkedList`

12.28

回忆第 10 章，在 Java 类库中含有接口 `java.util.List`。这个接口与我们的 `ListInterface` 类似，但它声明了更多的方法。另外，有些方法有不同的名字或规范说明，线性表项的位置从 0 开始而不是从 1 开始。第 10 章的段 10.14 概括了这些不同。

同一个包 `java.util` 含有类 `LinkedList`。这个类实现了接口 `List` 以及第 7 章描述的接口 `Queue` 和 `Deque`。所以 `LinkedList` 比接口 `List` 定义了更多的方法。进一步，你可以使用类 `LinkedList` 来实现 ADT 队列、双端队列或线性表。

## 本章小结

- 当链表仅有头引用时，以下叙述成立：
  - ◆ 在链表表头添加结点是特例。
  - ◆ 从链表中删除首结点是特例。
  - ◆ 在链表表尾添加或删除结点需要遍历整个链表。
  - ◆ 在链表中添加结点最多需要改变两个引用。
  - ◆ 从链表中删除结点最多需要改变两个引用。
- 当链表有头引用和尾引用时，以下叙述成立：

- ◆ 在空链表中添加结点是特例。
- ◆ 在链表表尾添加结点是特例，但不需要遍历。
- ◆ 删除链表的最后结点是特例。
- 除指向首结点的引用外，维护指向链表中最后结点的引用，当在链表表尾添加结点时可以消除遍历。所以，当链表有头引用和尾引用时，在线性表表尾的添加，比仅有头引用时更快。因此，我们使用有头引用和尾引用的链表来实现 ADT 线性表。

## 练习

1. 给类 `LList` 添加一个构造方法，从给定的对象数组创建线性表。至少考虑两种不同的方法实现这样一个构造方法。哪种方式需要做得更少？
2. 写一个测试类 `LList` 的程序。
3. 考虑段 12.11 给出的将项添加在线性表给定位置的 `add` 方法的定义。使用下面的代码替换情形 1 中执行的语句：

```
if (isEmpty() || (givenPosition == 1)) // Case 1
{
 firstNode = newNode;
 newNode.setNextNode(firstNode);
}
```

- a. 修改后的 `LList` 类的客户程序中的下列语句将显示什么？

```
ListInterface<String> myList = new LList<>();
myList.add(1, "30");
myList.add(2, "40");
myList.add(3, "50");
myList.add(1, "10");
myList.add(5, "60");
myList.add(2, "20");
int numberofEntries = myList.getLength();
for (int position = 1; position <= numberofEntries; position++)
 System.out.print(myList.getEntry(position) + " ");
```

- b. 修改 `add` 方法会影响到 `LList` 中哪些方法（如果有）的运行？为什么？
4. 假定你需要 ADT 线性表的一个操作，它将项的数组添加到线性表的表尾。方法头可能是这样的：

```
public void addAll(T[] items)
```

为类 `LList` 实现这个方法。

5. 为类 `LList` 定义方法 `getPosition`，如第 11 章练习 2 所描述的。比较这个方法与类 `AList` 中定义的 `getPosition` 的执行时间。
6. 为类 `LList` 实现 `equals` 方法，当线性表中的项与第二个线性表中的项相等时返回真。
7. 重做第 11 章的练习 10，但使用 `LList` 类替换 `AList`。
8. 假定线性表含有 `Comparable` 对象。实现方法，返回小于给定项的项组成的新线性表。方法头可能是这样的：

```
public ListInterface<T> getAllLessThan(Comparable<T> anObject)
```

为类 `LList` 实现这个方法。确保你的方法不会影响到原始线性表的状态。

9. 为类 `LList` 定义第 11 章练习 3 描述的方法 `remove`。
10. 重做前一个练习，但从线性表中删除 `anObject` 的所有出现。
11. 为类 `LList` 定义第 11 章练习 4 描述的方法 `moveToEnd`。
12. 为类 `LList` 实现 `replace` 方法，返回一个布尔值。

13. 假定线性表含有 Comparable 对象。为类 LList 定义第 11 章练习 7 描述的方法 getMin 和 removeMin。
14. 考虑第 11 章给出的 AList 的 arrayList 实例。让线性表的初始大小为 10。且 LList 的实例称为 chainList。
- 向 arrayList 中添加了 145 个项后，底层数组有多大？
  - 再向 arrayList 中添加 20 个项后，底层数组有多大？
  - 向 chainList 中添加 145 个项后，链表中有多少个结点？
  - 再向 chainList 中添加 20 个项后，链表中有多少个结点？
  - 链表中的每个结点都有两个引用，所以含  $n$  个结点的链表有  $2n$  个引用。而大小为  $n$  的数组有  $n$  个引用。计算 a ~ d 中所描述的各种情形下引用的数目。
  - arrayList 使用的引用数何时比 chainList 少？
  - chainList 使用的引用数何时比 arrayList 少？

15. 第 3 章练习 12 描述的双向链表中的结点含有指向前一个结点和下一个结点的引用。在第 3 章中，双向链表仅有一个头引用，但它也能既有头引用也有尾引用，如图 12-12 所示。

列出将新结点添加到双向链表中所需的步骤，分别考虑以下情况：

- 新结点是链表中的第一个结点
- 新结点是链表中最后一个结点
- 新结点位于链表中两个已有结点之间

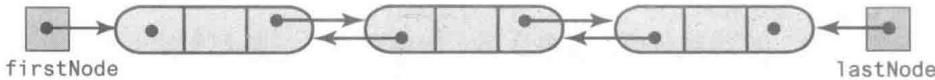


图 12-12 用于练习 15 和练习 16 及项目 8 的双向链表

16. 列出从图 12-12 所示的双向链表中删除一个结点时所需的步骤，分别考虑以下情况：

- 删除链表中第一个结点
- 删除链表中最后一个结点
- 删除位于链表中两个已有结点之间的结点

## 项目

- 写一个程序，充分测试类 LListWithTail。包括回答第 10 章练习 1 和练习 2 时的方法。
- 第 3 章程序清单 3-5 表明，类 Node 是包的一部分。创建另一个含有 Node、LList 和 ListInterface 的包。修改 LList，让其使用 Node 的这个版本。
- 创建 Java 接口，为线性表声明下列额外的方法：

```
/** Adds a new entry to the beginning of this list. */
public void addFirst(T newEntry)

/** Adds a new entry to the end of this list. */
public void addLast(T newEntry)

/** Removes and returns the first entry in this list. */
public T removeFirst()

/** Removes and returns the last entry in this list. */
public T removeLast()

/** Returns the first entry in this list. */
public T getFirst()

/** Returns the last entry in this list. */
public T getLast()

/** Moves the first entry in this list to the end of the list. */
public void moveToEnd()
```

然后从这个接口和 `ListInterface` 派生，定义 `DoubleEndedListInterface`。写一个实现 `DoubleEndedListInterface` 的类。使用有头引用和尾引用的结点链表表示线性表项。写一个程序充分测试你的类。

4. 重做前一个项目，但不使用尾引用。
5. 在链表中添加或删除结点，当操作在链表表头时需要特殊处理。为消除特殊情形，可以在链表表头添加一个哑结点（dummy node）。哑结点总存在，但不含有线性表项。这样链表永远不为空，故头引用永远不会是 `null`，哪怕线性表为空。修改本章提出的类 `LList`，在链表中增加一个哑结点。
6. 在循环链表中，最后一个结点指向首结点。通常，仅需维护一个外部引用——指向最后一个结点，因为从最后一个结点很容易找到首结点。第 8 章图 8-12 说明了这样一个链表。  
修改本章提出的类 `LList`，使用循环链表和尾引用。
7. 实现如第 1 章项目 3 描述的 ADT 环的类 `Ring`。将环表示为结点链表。考虑使用前一个项目中描述的循环链表。
8. 实现并测试使用图 12-12 所示的双向链表表示线性表项的线性表类。使用第 3 章练习 12 要求你定义的结点的内部类，但包含设置和获取方法。
9. 可以在双向链表表头添加一个哑结点，如项目 5 所描述的。修改前一个项目描述的 ADT 线性表的实现，在链表表头添加一个哑结点。
10. 定义链式实现接口 `FixedSizeListInterface` 的类，如第 11 章项目 9 所描述的。
11. 使用类 `LList` 而不是类 `AList`，重做第 11 章的任一项目。哪个实现的时间效率更好？为什么？

# 迭代器

先修章节：Java 插曲 2、第 10 章

迭代器是一个能遍历数据集合的对象。在遍历过程中，可以查看数据项、修改数据项、添加数据项及删除数据项。在 Java 类库中含有两个接口 `Iterator` 和 `ListIterator`，规范说明了用于迭代器的方法。当将这些迭代器方法添加为 ADT 的操作时，应该将它们实现为单独的类，来与 ADT 进行交互。这个迭代器类可以在 ADT 之外，或者隐藏在它的实现中。本插曲及第 13 章中将研究这两种方法。

## 什么是迭代器

J4.1 你如何数本页中的行数？当你数行时可以用手指点着每一行。手指帮忙记录在本页中数到的位置。如果数到某行时暂停了一下，则手指会停在当前行，可能会有前一行和下一行。如果将本页看作行的线性表，则数行的过程即是遍历了这个线性表。

迭代器（iterator）能让你从第一项开始，一步步地经过，或遍历（traverse）一个数据集合，例如线性表。在一次完整的遍历或迭代（iteration）过程中，每个数据项都被访问一次。通过重复地要求迭代器为你提供指向集合中下一项的引用，来控制迭代过程。还可以在遍历时修改集合，添加、删除或简单地修改其中的项。

因为已经写过循环语句，所以你已经熟悉了迭代。例如，如果 `nameList` 是字符串线性表，则可以写下面的 `for` 循环来显示整个线性表：

```
int listSize = nameList.getLength();
for (int position = 1; position <= listSize; position++)
 System.out.println(nameList.getEntry(position));
```

这个是循环遍历，或迭代（iterate）线性表中的各项。不只是简单显示每个项，我们可能还要做其他的事情。

J4.2 注意到，前面这个循环是客户层的，因为它用到了 ADT 操作 `getEntry` 来访问线性表。对于基于数组实现的线性表，`getEntry` 可以直接快速地得到想要的数组项。但如果用结点链表表示线性表项，`getEntry` 必须从一个结点移向另一个结点，直到到达想要的结点。例如，为获取线性表中的第  $n$  项，`getEntry` 应该从链表的首结点开始，然后移向第二个结点、第三个结点，以此类推，直到到达第  $n$  个结点。在循环的下一次重复中，`getEntry(n+1)` 将再从链表的首结点开始，一步步地从一个结点到下一个结点，直到到达第  $n+1$  个结点，从而获取线性表的第  $n+1$  项。也就是说，`getEntry(n+1)` 不能从 `getEntry(n)` 查找链表时离开的地方开始。这很浪费时间。

迭代是一个常用的操作，可以将它视作 ADT 线性表的一部分。这样做，比在客户层实现的效率要高。注意，ADT 线性表的 `toArray` 操作执行的就是遍历。但它是由 ADT 控制的遍历。客户可以调用 `toArray`，但一旦开始就不能控制遍历。

但是 `toArray` 只返回线性表的项。在遍历项时要是想对项做其他的操作该怎么办呢？

我们不想在每次用另一种方式使用迭代时都给 ADT 增加新的操作。我们需要为客户提供一种方法，让其能一步步经过数据集合以获取或修改项。遍历应该能记录下自己处理的过程，即它能知道处在集合中的什么位置，及是否已经访问了每个项。迭代器提供的是这样的一个遍历。



### 注：迭代器

迭代器是一个对象，它能一步步地经过，或遍历一个数据集合。迭代器在遍历或迭代过程中，能记录下自己处理的过程。它能告诉你下一项是否存在，如果存在，能返回指向它的引用。一个迭代周期内，每个数据项都被访问一次。

Java 类库中的包 `java.util` 含有两个标准接口——`Iterator` 和 `ListIterator`——它们规范说明了适用于迭代器的方法。下面先来看看 `Iterator` 接口。

## 接口 `Iterator`

与我们讨论过的大多数接口一样，程序清单 JI4-1 中给出的 `Iterator` 规范说明了用泛型来表示迭代时处理的项的数据类型。接口中只规范说明了一个迭代器可以具有的 3 个方法——`hasNext`、`next` 和 `remove`。这些方法能让你从头开始遍历数据集合。J4.3

**程序清单 JI4-1** Java 的接口 `java.util.Iterator`

```

1 package java.util;
2 public interface Iterator<T>
3 {
4 /** Detects whether this iterator has completed its traversal
5 * and gone beyond the last entry in the collection of data.
6 * @return True if the iterator has another entry to return. */
7 public boolean hasNext();
8
9 /** Retrieves the next entry in the collection and
10 * advances this iterator by one position.
11 * @return A reference to the next entry in the iteration,
12 * if one exists.
13 * @throws NoSuchElementException if the iterator had reached the
14 * end already, that is, if hasNext() is false. */
15 public T next();
16
17 /** Removes from the collection of data the last entry that
18 * next() returned. A subsequent call to next() will behave
19 * as it would have before the removal.
20 * Precondition: next() has been called, and remove() has not
21 * been called since then. The collection has not been altered
22 * during the iteration except by calls to this method.
23 * @throws IllegalStateException if next() has not been called, or
24 * if remove() was called already after the last call to next().
25 * @throws UnsupportedOperationException if the iterator does
26 * not permit a remove operation. */
27 public void remove(); // Optional method
28 } // end Iterator

```

迭代器标记它在集合中的当前位置，很像是你的手指指向线性表中的一个项或是本页中的一行。但是，在 Java 中，迭代器的位置不在项上，而是在集合的第一个项之前，或是两项之间，或是最后一项之后，如图 JI4-1 所示。对于含  $n$  项的集合，游标有  $n+1$  个可能的位置。迭代的下一项（`next entry`）是迭代器游标（cursor）位置右侧的项。方法 `hasNext` 查看J4.4

下一项是否存在，且据此返回真或假。



图 JI4-1 迭代器游标在集合中的可能位置

如果 `hasNext` 返回真，则方法 `next` 将迭代器的游标移过下一项，并返回指向该项的引用，如图 JI4-2a 和 JI4-2b 所示。重复调用 `next` 可以在集合中进行遍历。迭代过程中，迭代器返回一项又一项。一旦 `next` 已经返回了集合中的最后一项，则后面再调用它都会引发 `NoSuchElementException`。

方法 `remove` 删除 `next` 刚刚返回的项，如图 JI4-2c 所示。ADT 线性表的 `remove` 操作与此不同，它删除线性表中指定位置的项。当实现 `Iterator` 接口时，不一定非要提供 `remove` 操作——它是可选的——但因为它出现在接口中，所以你确实需要定义方法 `remove`。如果客户调用这样一个方法，它应该抛出异常 `UnsupportedOperationException`。



**注：**Java 的接口 `java.util.Iterator` 规范说明了 3 个方法：`hasNext`、`next` 和 `remove`。方法 `hasNext` 查看迭代器是否有下一项要返回。如果有，则 `next` 返回指向它的引用。方法 `remove` 可以删除调用 `next` 时最后返回的项，或是如果不允许迭代器进行删除，只需抛出 `UnsupportedOperationException`。



**程序设计技巧：**接口 `Iterator` 中提到的所有异常都是运行时异常，故不需要在任何方法的头部写 `throws` 子句。另外，当调用这些方法时也不必写 `try` 和 `catch` 块。但是，必须从包 `java.util` 引入 `NoSuchElementException`。其他的异常在 `java.lang` 中，所以对它们不需要使用 `import` 语句。

## 接口 `Iterable`

J4.5

对于一个给定集合，可能有不同的方法得到迭代器。一种方法是，对于集合本身，创建并给出这样一个迭代器。实现标准接口 `java.lang.Iterable`——程序清单 JI4-2 所示——的类可以做到这一点。这个接口仅声明了一个方法 `iterator`，它返回一个符合 `Iterator` 接口的迭代器。下面的示例中使用的就是这个方法。

### 程序清单 JI4-2 接口 `java.lang.Iterable`

```

1 package java.lang;
2 public interface Iterable<T>
3 {
4 /** @return An iterator for a collection of objects of type T. */
5 Iterator<T> iterator();
6 } // end Iterable

```



图 JI4-2 调用 `next` 且随后调用 `remove` 对集合迭代器的效果

## 使用接口 Iterator

使用迭代器的一些细节，取决于实现迭代器方法的方式。下一章将探讨这些方式。现在，我们假定，实现 ADT 的类有一个方法，它能返回用于 ADT 项的迭代器，接下来我们只关注于接口 `Iterator` 中的方法应有怎样的行为。

 **示例。**下面来看一个示例，看看接口 `Iterator` 中的方法 `hasNext` 和 `next`，如何处理 J4.6  
ADT 线性表。假定我们的 `ListInterface` 派生于接口 `Iterable`，且类 `MyList` 实现了 `ListInterface`。下列语句创建了一个名字线性表，其中的项是简单的字符串：

```
ListInterface<String> nameList = new MyList<>();
nameList.add("Jamie");
nameList.add("Joey");
nameList.add("Rachel");
```

此时，`nameList` 中含有字符串

Jamie

Joey

Rachel

要得到 `nameList` 的迭代器，可以调用 `nameList` 的方法 `iterator`，如下所示：

```
Iterator<String> nameIterator = nameList.iterator();
```

迭代器 `nameIterator` 位于线性表的第一项之前。图 JI4-3 图示了展示迭代器方法的一系列事件。

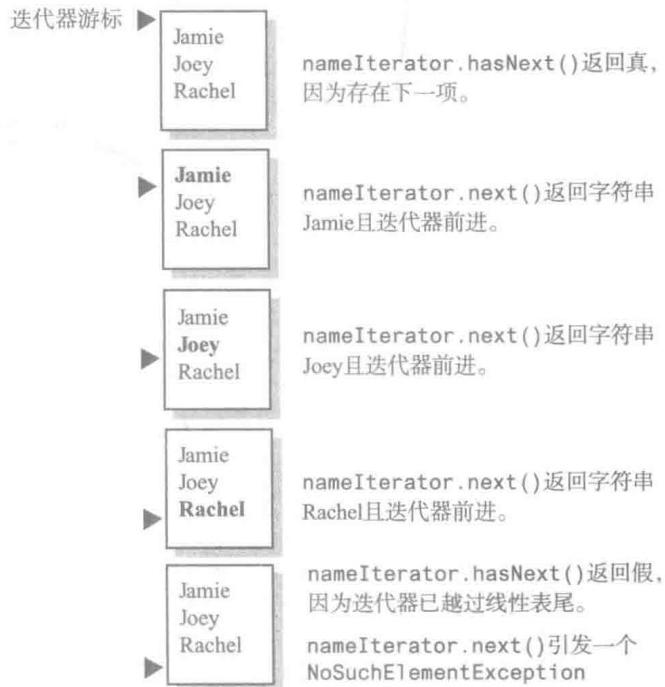


图 JI4-3 迭代器方法 `hasNext` 和 `next` 作用于线性表的效果

 **示例。**可以使用迭代器来显示线性表中的所有项。下列语句显示线性表 `nameList` 中 J4.7  
的字符串，一行一个：

```
Iterator<String> nameIterator = nameList.iterator();
while (nameIterator.hasNext())
 System.out.println(nameIterator.next());
```

先调用 `nameList` 的方法 `iterator`, 创建迭代器对象。得到的迭代器 `nameIterator` 恰位于线性表第一项之前。所以, `nameIterator.next()` 将返回第一项, 且迭代器前进到线性表的第二项之前。如果 `hasNext` 返回真, 则 `next` 返回线性表中的下一项且迭代器前进。所以能获取线性表中的每一项并显示出来。

J4.8  **示例。** 接口 `Iterator` 提供了从数据集中删除项的操作。这个项是最后一次调用方法 `next` 时返回的项。所以在调用 `remove` 之前必须先调用 `next`。

假设 `nameList` 含有字符串 `Andy`、`Brittany` 和 `Chris`, 且 `nameIterator` 由前一个示例定义。图 JI4-4 图示了在迭代 `nameList` 的过程中, `Iterator` 的 `remove` 方法的行为。



图 JI4-4 迭代器方法 `next` 和 `remove` 作用于线性表上的效果

J4.9  **示例。** 在调用 `remove` 之前要先调用 `next` 的这个要求, 使得在两种情况下会引发 `IllegalStateException`。若 `nameList` 如前一个示例定义的那样, 则语句

```
Iterator<String> nameIterator = namelist.iterator();
nameIterator.hasNext();
nameIterator.remove();
```

会引发 `IllegalStateException`, 因为在调用 `remove` 之前没有调用 `next`。类似地, 换成如下的语句

```
nameIterator.next();
nameIterator.remove();
nameIterator.remove();
```

第二个 `remove` 语句会引发 `IllegalStateException`, 因为从最近一次调用 `next` 后, `remove` 已被调用过了。



**学习问题 1** 假定 `nameList` 中含有名字 Jamie、Joey 和 Rachel, 如段 J4.6 一样。下列 Java 语句会得到什么输出?

```
Iterator<String> nameIterator = namelist.iterator();
nameIterator.next();
nameIterator.next();
nameIterator.remove();
System.out.println(nameIterator.hasNext());
System.out.println(nameIterator.next());
```



**学习问题 2** 如果 `nameList` 中至少含有 3 个字符串, 且 `nameIterator` 如学习问题 1 中所定义的, 写出 Java 语句, 显示线性表的第 3 项。

**学习问题 3** 给定如学习问题 2 中的 `nameList` 和 `nameIterator`, 写出语句, 显示线性表中的偶数项。即显示第二项、第四项, 等等。

**学习问题 4** 给定学习问题 2 中所描述的 `nameList` 和 `nameIterator`, 写出语句, 删除线性表中的所有项。

**多个迭代器。** 尽管前一个示例展示的是一个迭代器遍历集合, 不过我们对同一个集合可以同时有多个迭代器。例如, 假定打印的一个名字列表有重复的名字, 也没有特殊的次序。用一根手指滑过这个名单来数名字的个数, 类似于线性表的一次迭代。现在假定, 你想统计每个名字在打印的名单中出现的次数。可以如下这样使用两根手指。用你左手的手指向名单表中的第一个名字。再用右手的手指指向这个表中的每个名字, 从第一个开始。当你用右手遍历名单时, 将每个名字与左手指向的名字进行比较。用这种方法, 可以统计名单中第一个名字出现的次数。现在左手手指移向名单中的下一个名字, 且用右手手指指向名单的开头。重复刚才这个过程, 来统计名单中出现的第二个名字的出现次数。用图 JI4-5 中的名字来试一试。(因为你左手将指向 Jane 3 次, 所以进行了无用的重复计算, 除非你很小心。我们稍后讨论这个细节。)

两根手指可以独立地遍历。它们很像是两个独立的遍历同一个线性表的迭代器, 如下一个示例中所见的。



**示例。** 现在来写代码, 统计图 JI4-5 所示的每个名字出现的次数。让 `nameIterator` 对应于图中你的左手。现在定义第二个迭代器 `countingIterator`, 它对应于你的右手。对左手指向的每个名字, 右手遍历整个名单, 来统计这个名字出现的次数。所

|    |        | 名单中Jane<br>出现的次数  | JI4.10 |
|----|--------|-------------------|--------|
| 左手 | Brad   | 在名单<br>中前进<br>的右手 | 0      |
|    | Jane   | 1                 |        |
|    | Bob    | 1                 |        |
|    | Jane   | 2                 |        |
|    | Bette  | 2                 |        |
|    | Brad   | 2                 |        |
|    | Jane   | 3                 |        |
|    | Brenda | 3                 |        |
|    |        | Jane出现3次          |        |

图 JI4-5 统计名字列表中 Jane 出现的次数

以，有下列嵌套的循环，假定 `nameList` 是线性表：

```
Iterator<String> nameIterator = namelist.iterator();
while (nameIterator.hasNext())
{
 String currentName = nameIterator.next();
 int nameCount = 0;
 Iterator<String> countingIterator = namelist.iterator();
 while (countingIterator.hasNext())
 {
 String nextName = countingIterator.next();
 if (currentName.equals(nextName))
 nameCount++;
 } // end while
 System.out.println(currentName + " occurs " + nameCount + " times.");
} // end while
```

要将 `countingIterator` 重置到线性表的开头，可以再次调用方法 `iterator`，因为接口 `Iterator` 中没有用来做这个事情的方法。

对于图 JI4-5 中所给的名字，这些语句得到下列输出：

```
Brad occurs 2 times.
Jane occurs 3 times.
Bob occurs 1 times.
Jane occurs 3 times.
Bette occurs 1 times.
Brad occurs 2 times.
Jane occurs 3 times.
Brenda occurs 1 times.
```

正如你所见，因为 `nameIterator`（你的左手）遇到 Brad 两次，遇到 Jane 3 次，所以内层循环进行了无意义的重复计算。例如，每次 `nameIterator` 遇到 Brad 时，我们都计算 Brad 出现了 2 次。

如果 `nameIterator` 提供一个删除操作，且如果我们允许破坏线性表，则通过如下这样修改 `if` 语句，可以删除重复的项——因此可以不必进行重复计算：

```
if (currentName.equals(nextName))
{
 nameCount++;
 if (nameCount > 1)
 countingIterator.remove();
} // end if
```

当 `nameCount` 大于 1 时，`nextName` 一定是迭代器 `countingIterator` 从线性表中已经获取的不止一次的名字。所以我们删除那个项，这样 `nameIterator` 就不会再遇到它。通过调用 `countingIterator.remove()` 来达到这个目的。然后，迭代器 `countingIterator` 继续处理下一项。

## Iterable 和 for-each 循环

**J4.12** 实现了接口 `Iterable` 的类，比其他没有实现这个接口的类有独特的优势：可以使用 `for-each` 循环，来遍历这种类的实例的对象。例如，假定 `nameList` 是刚刚创建的线性表类的实例，且类实现了 `Iterable`。现在给这个空线性表添加 4 个字符串，如下：

```
nameList.add("Joe");
nameList.add("Jess");
```

```
nameList.add("Josh");
nameList.add("Jen");
```

则语句

```
for (String name : nameList)
 System.out.print(name + " ");
System.out.println();
```

会得到下列输出：

```
Joe Jess Josh Jen
```

## 接口 ListIterator

Java 类库在包 `java.util` 中提供了用于迭代器的第二个接口——`ListIterator`。这个类型的迭代器能让你双向遍历线性表、得到迭代器当前位置、遍历过程中修改线性表。除了接口 `Iterator` 中规范说明的 `hasNext`、`next` 和 `remove` 这 3 个方法外，`ListIterator` 还含有像 `hasPrevious`、`previous`、`add` 和 `set` 这样的方法。与 `Iterator` 一样，`ListIterator` 将迭代器的游标定位在集合第一项之前、两个项之间或是最后一项之后，如图 JI4-1 所示。

J4.13

我们先来看程序清单 JI4-3 中列出的接口 `ListIterator`

**程序清单 JI4-3** Java 的接口 `java.util.ListIterator`

```
1 package java.util;
2 public interface ListIterator<T> extends Iterator<T>
3 {
4 /** Detects whether this iterator has gone beyond the last
5 * entry in the list.
6 * @return True if the iterator has another entry to return when
7 * traversing the list forward; otherwise returns false. */
8 public boolean hasNext();
9
10 /** Retrieves the next entry in the list and
11 * advances this iterator by one position.
12 * @return A reference to the next entry in the iteration,
13 * if one exists.
14 * @throws NoSuchElementException if the iterator is at the end,
15 * that is, if hasNext() is false. */
16 public T next();
17
18 /** Removes from the list the last entry that either next()
19 * or previous() has returned.
20 * Precondition: next() or previous() has been called, but the
21 * iterator's remove() or add() method has not been called
22 * since then. That is, you can call remove only once per
23 * call to next() or previous(). The list has not been altered
24 * during the iteration except by calls to the iterator's
25 * remove(), add(), or set() methods.
26 * @throws IllegalStateException if next() or previous() has not
27 * been called, or if remove() or add() has been called
28 * already after the last call to next() or previous().
29 * @throws UnsupportedOperationException if the iterator does not
30 * permit a remove operation. */
31 public void remove(); // Optional method
32
33 // The previous three methods are in the interface Iterator; they are
34 // duplicated here for reference and to show new behavior for remove.
35}
```

```

36 /** Detects whether this iterator has gone before the first
37 * entry in the list.
38 * @return True if the iterator has another entry to visit when
39 * traversing the list backward; otherwise returns false. */
40 public boolean hasPrevious();
41
42 /** Retrieves the previous entry in the list and moves this
43 * iterator back by one position.
44 * @return A reference to the previous entry in the iteration, if
45 * one exists.
46 * @throws NoSuchElementException if the iterator has no previous
47 * entry, that is, if hasPrevious() is false. */
48 public T previous();
49
50 /** Gets the index of the next entry.
51 * @return The index of the list entry that a subsequent call to
52 * next() would return. If next() would not return an entry
53 * because the iterator is at the end of the list, returns
54 * the size of the list. Note that the iterator numbers
55 * the list entries from 0 instead of 1. */
56 public int nextIndex();
57
58 /** Gets the index of the previous entry.
59 * @return The index of the list entry that a subsequent call to
60 * previous() would return. If previous() would not return
61 * an entry because the iterator is at the beginning of the
62 * list, returns -1. Note that the iterator numbers the
63 * list entries from 0 instead of 1. */
64 public int previousIndex();
65
66 /** Adds an entry to the list just before the entry, if any,
67 * that next() would have returned before the addition. This
68 * addition is just after the entry, if any, that previous()
69 * would have returned. After the addition, a call to
70 * previous() will return the new entry, but a call to next()
71 * will behave as it would have before the addition.
72 * Further, the addition increases by 1 the values that
73 * nextIndex() and previousIndex() will return.
74 * @param newEntry An object to be added to the list.
75 * @throws ClassCastException if the class of newEntry prevents the
76 * addition to the list.
77 * @throws IllegalArgumentException if some other aspect of
78 * newEntry prevents the addition to the list.
79 * @throws UnsupportedOperationException if the iterator does not
80 * permit an add operation. */
81 public void add(T newEntry); // Optional method
82
83 /** Replaces the last entry in the list that either next()
84 * or previous() has returned.
85 * Precondition: next() or previous() has been called, but the
86 * iterator's remove() or add() method has not been called since then.
87 * @param newEntry An object that is the replacement entry.
88 * @throws ClassCastException if the class of newEntry prevents the
89 * addition to the list.
90 * @throws IllegalArgumentException if some other aspect of newEntry
91 * prevents the addition to the list.
92 * @throws IllegalStateException if next() or previous() has not
93 * been called, or if remove() or add() has been called
94 * already after the last call to next() or previous().
95 * @throws UnsupportedOperationException if the iterator does not
96 * permit a set operation. */
97 public void set(T newEntry); // Optional method
98 } // end ListIterator

```

观察。注意到，`ListIterator` 派生于 `Iterator`。所以 `ListIterator` 应该含有 `Iterator` 中的方法 `hasNext`、`next` 和 `remove`，即使没有明确地写出它们。我们这样写是为了供你参考，并指出 `remove` 的附加功能。J4.14

方法 `remove`、`add` 和 `set` 都是可选的，从这一方面来说，你可以选择去掉其中的一个或多个。不过如果是那样，去掉的每个操作都必须实现为，当客户调用这个操作时要抛出异常 `UnsupportedOperationException`。不支持 `remove`、`add` 和 `set` 的 `ListIterator` 类型的迭代器仍是有用的，因为它能让你双向遍历线性表。如果没有这些操作，它的实现还更容易。

段 J4.4 中为接口 `Iterator` 给出的程序设计技巧也适用于这里。我们换作 `ListInterface` 再重说一遍。

**!** 程序设计技巧：接口 `ListInterface` 中提到的所有异常都是运行时异常，故不需要在任何方法的头部写 `throws` 子句。另外，当调用这些方法时也不必写 `try` 和 `catch` 块。但是，必须从包 `java.util` 引入 `NoSuchElementException`。其他的异常在 `java.lang` 中，所以对它们不需要使用 `import` 语句。J4.14

下一项。回忆方法 `hasNext` 查看迭代器位置的后面是否存在下一项。如果存在，则 `next` 将返回指向它的引用，且迭代器游标前进一个位置，如图 JI4-2 所示。重复调用 `next` 可以一步步地经过线性表。到目前为止所学的内容与本插曲开头的接口 `Iterator` 毫无二致。J4.15

前一项。`ListIterator` 还提供了访问迭代器位置之前的项——即前一项——的能力。方法 `hasPrevious` 查看前一项是否存在。如果存在，则方法 `previous` 返回指向它的引用且迭代器游标回退一个位置。图 JI4-6 显示 `previous` 作用于线性表的效果。混合调用 `previous` 和 `next` 方法，能让你在线性表中来回移动。如果调用 `next`，然后再调用 `previous`，则两个方法返回的是同一项。与 `next` 一样，当已完成线性表的遍历后再调用 `previous`，它会抛出异常。J4.16

**♪** 注：`ListIterator` 类型的迭代器的游标位置总是位于 `previous` 返回的项及 `next` 返回的项的中间。



图 JI4-6 在线性表上调用 `previous` 的效果

当前项和前一项的下标。如图 JI4-7 所示，方法 `nextIndex` 和 `previousIndex` 分别返回随后调用 `next` 或 `previous` 应返回的项的下标。注意到，迭代器将线性表项从 0 开始记数，而不是像 ADT 线性表操作中那样从 1 开始计数。如果因为迭代器处在线性表尾而在调用 `next` 时抛出一个异常，则 `nextIndex` 将返回线性表的大小。类似地，如果因为迭代器处在线性表头而在调用 `previous` 时抛出一个异常，则 `previousIndex` 将返回 -1。J4.17

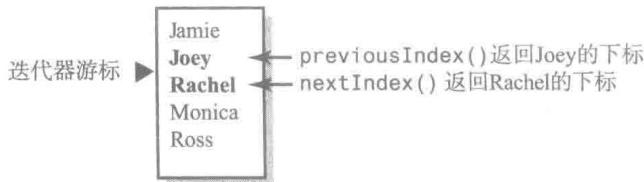


图 JI4-7 方法 nextIndex 和 previousIndex 返回的下标



**注:** 接口 ListIterator 规范说明了 9 个方法，包括 Iterator 规范说明的 3 个方法。它们是 hasNext、hasPrevious、next、previous、nextIndex、previousIndex、add、remove 和 set。

## 再论接口 List

J4.18

第 10 章段 10.14 中描述的接口 `java.util.List` 派生于接口 `Iterable`，所以它有方法 `iterator`。另外，`List` 声明了下述与迭代器相关的方法：

```
public ListIterator<T> listIterator(int index);
public ListIterator<T> listIterator();
```

每个 `listIterator` 方法都返回一个迭代器，迭代器中的方法都已在接口 `ListIterator` 中规范说明。由第一个 `listIterator` 返回的迭代器，它的开始项是由 `index` 指示的线性表项，其中 0 表示线性表的第一项。`listIterator` 方法的第二个版本与 `listIterator(0)` 的作用相同。

因为 `java.util` 包中的标准类 `ArrayList`、`LinkedList` 和 `Vector` 都实现了接口 `List`，所以它们除有 `iterator` 方法外，都还有这两个 `listIterator` 方法。

## 使用接口 ListIterator

J4.19



**示例：遍历。** 现在来看操作当前项和前一项的方法示例，然后用这个示例来说明接口中其他的方法。有下列假定：

- 线性表 `nameList` 是 `java.util.List` 类型的，含有下列名字：

Jess  
Jim  
Josh

- 迭代器 `traverse` 定义如下

```
ListIterator<String> traverse = nameList.listIterator();
```

且含有操作 `add`、`remove` 和 `set`。

因为 `traverse` 位于线性表头，故 Java 语句

```
System.out.println("nextIndex " + traverse.nextIndex());
System.out.println("hasNext " + traverse.hasNext());
System.out.println("previousIndex " + traverse.previousIndex());
System.out.println("hasPrevious " + traverse.hasPrevious());
```

得到下列输出

```
nextIndex 0
hasNext true
previousIndex -1
hasPrevious false
```

然后，如果执行语句

```
System.out.println("next " + traverse.next());
System.out.println("nextIndex " + traverse.nextIndex());
System.out.println("hasNext " + traverse.hasNext());
```

则输出是

```
next Jess
nextIndex 1
hasNext true
```

最后，语句

```
System.out.println("previousIndex " + traverse.previousIndex());
System.out.println("hasPrevious " + traverse.hasPrevious());
System.out.println("previous " + traverse.previous());
System.out.println("nextIndex " + traverse.nextIndex());
System.out.println("hasNext " + traverse.hasNext());
System.out.println("next " + traverse.next());
```

得到输出

```
previousIndex 0
hasPrevious true
previous Jess
nextIndex 0
hasNext true
next Jess
```

 学习问题 5 假定 `traverse` 是前一段中定义的迭代器，但 `nameList` 的内容未知。写 Java 语句，按反序显示 `nameList` 中的名字，从表尾开始。



示例：方法 `set`。方法 `set` 替换 `next` 或是 `previous` 刚返回的项。前一段的最后，`next` 刚刚返回 `Jess`，所以

J4.20

```
traverse.set("Jen");
```

将用 `Jen` 替换 `Jess`。因为 `Jess` 是线性表的第一项，所以线性表现在是：

Jen  
Jim  
Josh

注意到，这个替换操作不影响迭代器在线性表中的位置。所以，调用一些方法，比如 `nextIndex` 和 `previousIndex`，都不会受到影响。这种情形下，因为迭代器位于 `Jen` 和 `Jim` 之间，所以 `nextIndex` 返回 `1`，而 `previousIndex` 返回 `0`。还要注意，我们可以再次调用 `set`，这次替换的是 `Jen`。



学习问题 6 如果迭代器位置位于前一个线性表的头两项之间，写 Java 语句，用 `Jon` 替换 `Josh`。

J4.21



**示例：方法 add。**方法 `add` 将一项插入线性表中恰在迭代器当前位置之前的地方。所以，如果在调用 `add` 之前调用了 `next` 或是 `previous`，则插入发生在 `next` 返回的项的前面或是 `previous` 返回的项的后面。注意，如果线性表为空，则 `add` 将新项插入为线性表中唯一的项。

如果迭代器的当前位置位于前面这个线性表的头两项之间，则语句

```
traverse.add("Ashley");
```

将 `Ashley` 插入线性表中 `Jim` 之前的地方——即下标为 1，或是作为线性表的第二项。添加后，线性表如下：

```
Jen
Ashley
Jim
Josh
```

在这个位置调用 `next` 将返回 `Jim`，因为没调用 `add` 时 `next` 应该返回 `Jim`。但如果不是调用 `next` 而是调用 `previous`，则返回新项 `Ashley`。另外，添加操作使得返回的 `nextIndex` 值和 `previousIndex` 值都加 1。所以添加操作后，`nextIndex` 将返回 2，而 `previousIndex` 将返回 1。



**学习问题 7** 如果迭代器位置位于 `Ashley` 和 `Jim` 之间，写 Java 语句，紧接在 `Jim` 的后面添加 `Miguel`。

J4.22



**示例：方法 remove。**方法 `remove` 的行为类似于我们在本插曲开头见过的接口 `Iterator` 中的 `remove`。但在接口 `ListIterator` 中，`remove` 除受 `next` 的影响外，也受方法 `previous` 的影响。所以，`remove` 删除最后一次调用 `next` 或 `previous` 时返回的线性表项。

如果线性表含有

```
Jen
Ashley
Jim
Josh
```

而迭代器 `traverse` 位于 `Ashley` 和 `Jim` 之间，则语句

```
traverse.previous();
traverse.remove();
```

将从线性表中删除 `Ashley`，因为 `previous` 返回 `Ashley`。迭代器位置仍在 `Jim` 之前。

注意到，如果既没有调用 `next` 方法也没有调用 `previous` 方法，或是从最后一次调用 `next` 或 `previous` 之后已经调用过 `remove` 或 `add`，则 `set` 和 `remove` 都会抛出 `IllegalStateException` 异常。如第 13 章所见，这个行为使得实现有些复杂。

# ADT 线性表的迭代器

先修章节：第 11 章、第 12 章、Java 插曲 4

## 目标

学习完本章后，应该能够

- 使用迭代器遍历或操作线性表
- 在 Java 中为线性表实现一个独立类迭代器及内部类迭代器
- 描述独立类迭代器和内部类迭代器的优缺点

从 Java 插曲 4 中已经知道，迭代器是一个对象，能让你遍历数据集合中的项。插曲中给出了使用迭代器的几个例子，并介绍了标准接口 `Iterator`、`ListIterator` 和 `Iterable`。本章，我们将为第 11 章和第 12 章定义的类 `AList` 和 `LList` 实现 `Iterator` 和 `ListIterator`。为此，我们将叙述定义迭代器的几种方法。

## 实现迭代器的方法

为 ADT 提供遍历操作的可能的方式是将这样的遍历操作定义为 ADT 操作，但它不是最理想的方式。例如，如果 `ListInterface` 继承于 `Iterator`，则线性表对象既会拥有线性表方法，又会拥有迭代器方法。不过这种实现方式虽然能提供高效的遍历，但它也有缺点，稍后会看到。13.1

更好的方法是在其自己的类内实现迭代器。一种方法是，这个类是公有的，且独立于所说的实现 ADT 的类。当然两个类必须以某种方式交互。我们将这样的一个迭代器类实例称为 **独立类迭代器** (*separate class iterator*)。另一种方法是，迭代器类可以是实现 ADT 的类的私有内部类。我们将这个内部类的实例称为 **内部类迭代器** (*inner class iterator*)。正如你将看到的，内部类迭代器通常更好一些。本章讨论这两种方法。

线性表的独立类迭代器和线性表的内部类迭代器，都是区别于线性表的对象。你调用这两个迭代器中的方法的方式是相同的。

## 独立类迭代器

假定 `nameList` 是字符串的线性表，是分别在第 11 章和第 12 章给出的 `AList` 或 `LList` 类的实例。如果公有类 `SeparateIterator` 实现了接口 `java.util.Iterator`，则我们可以为 `nameList` 创建一个迭代器，如下：13.2

```
Iterator<String> nameIterator = new SeparateIterator<>(nameList);
```

`SeparateIterator` 的构造方法将迭代器 `nameIterator` 与线性表 `nameList` 相关联，且将迭代器定位在线性表第一项之前。因为 `nameIterator` 是不同于线性表类的迭代器类的一个实例，所以它是一个独立类迭代器。可以像 Java 插曲 4 的段 J4.6 和段 J4.8 中所做的那样，来使用 `nameIterator`，因为它拥有接口 `Iterator` 所声明的方法。

下面来定义类 SeparateIterator。

**13.3** 类 SeparateIterator 的框架。为了通过调用类 SeparateIterator 的构造方法，将这个类的实例与一个已有的线性表相关联，SeparateIterator 需要一个线性表引用的数据域。正如在程序清单 13-1 中将看到的，构造方法将这个引用赋给数据域 list。而且注意到，定义 SeparateIterator 时不依赖于线性表的具体实现，比如是 AList 或是 LList，而是将数据域 list 定义为 ListInterface 的实例。

### 程序清单 13-1 类 SeparateIterator 的框架

```

1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3 public class SeparateIterator<T> implements Iterator<T>
4 {
5 private ListInterface<T> list;
6 private int nextPosition; // Position of entry last returned by next()
7 private boolean wasNextCalled; // Needed by remove
8
9 public SeparateIterator(ListInterface<T> myList)
10 {
11 list = myList;
12 nextPosition = 0;
13 wasNextCalled = false;
14 } // end constructor
15
16 < Implementations of the methods hasNext, next, and remove go here. >
17 .
18 } // end SeparateIterator

```

除了将迭代器与所讨论的线性表进行关联外，构造方法要将迭代器初始化，所以它开始时位于线性表的第一项。为此，类有另一个数据域 nextPosition，它记录我们迭代到的位置。这个域是一个整数，它是 next 方法最后返回的线性表中项的位置。方便起见将这个域初始化为 0。

是否提供一个带删除操作的迭代器，随你意而定。此处我们定义的迭代器是带有这个操作的，因为前一个例子用到了。这个需求使得我们的类有些复杂，因为客户必须在每次调用 remove 之前调用方法 next。这个要求不简简单单的只是一个前置条件。如果不符合，remove 方法必须抛出一个异常。所以，我们还需要另外一个数据域——布尔标志——以确保 remove 去检查是否已经调用了 next。将这个数据域称为 wasNextCalled。构造方法将这个域初始化为假。

**13.4** 方法 hasNext。类 SeparateIterator 不能直接访问实现线性表的类的私有数据域。它是线性表的客户，所以只能使用线性表的 ADT 操作来处理线性表。图 13-1 显示了一个独立类迭代器，它带一个指向线性表的引用，但它不知道线性表的实现细节。实现迭代器方法时将用到 ListInterface 中规范说明的方法。这样的实现相当简单，但一般来讲，比起内部类迭代器的实现，其运行时间更长。例如，方法 hasNext 调用线性表的 getLength 方法：

```

public boolean hasNext()
{
 return nextPosition < list.getLength();
} // end hasNext

```



学习问题 1 当线性表为空时，方法 hasNext 返回什么？为什么？

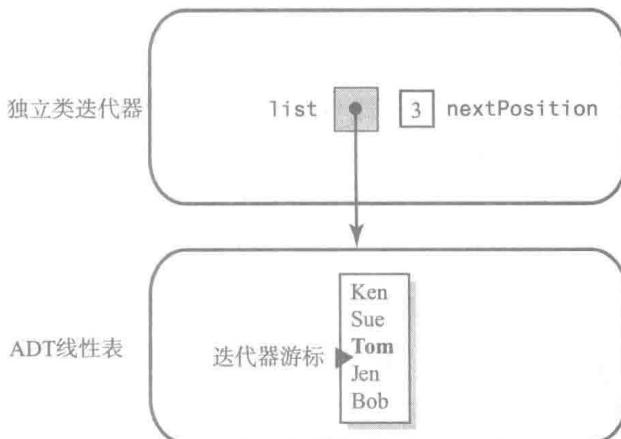


图 13-1 带指向 ADT 的引用、带迭代过程中的位置指示器但不知道 ADT 实现细节的独立类迭代器

**方法 next。**只要迭代没达到最后位置——即只要 `hasNext` 返回真——方法 `next` 就通过调用线性表的 `getEntry` 方法得到迭代的下一项。但如果迭代器已到达最后位置，则 `next` 方法抛出一个异常。

```
public T next()
{
 if (hasNext())
 {
 wasNextCalled = true;
 nextPosition++;
 return list.getEntry(nextPosition);
 }
 else
 throw new NoSuchElementException("Illegal call to next(); " +
 "iterator is after end of list.");
} // end next
```

因为 `nextPosition` 从 0 开始，所以我们必须在将它传给 `getEntry` 之前先加 1。在迭代器之前做这一步是必需的。注意到，我们还将域 `wasNextCalled` 设置为真，为的是方法 `remove` 能知道已经调用了 `next`。

**学习问题 2** 方法 `next` 要做的具体工作，取决于最终使用的 ADT 线性表的实现方式。在线性表的哪种实现方式下，基于数组还是链式，`next` 需要的运行时间最多？为什么？

**方法 remove。**迭代器的方法 `remove` 从线性表中删除最近一次调用 `next` 时返回的项。如果没有调用过 `next` 方法，或者从最后一次调用 `next` 方法后如果已经调用过 `remove` 方法，则 `remove` 方法抛出异常 `IllegalStateException`。实现方法的这个功能时要用到类的数据域 `wasNextCalled`。如果这个域的值为真，则我们知道已经调用过 `next` 方法了。然后将该数据域设置为假，保证随后调用 `remove` 方法时需要再次调用 `next` 方法。

数据域 `nextPosition` 是刚刚由 `next` 返回的项的位置，所以这是要删除项的位置。因此，将它传给线性表的 `remove` 方法。因为随后调用 `next` 时的动作必须与删除之前是一样的，故必须让 `nextPosition` 减 1。

图 13-2 显示的是一个线性表，还包括刚要调用 `next` 之前、刚调用完 `next` 之后但调用 `remove` 之前，及刚调用完 `remove` 之后的域 `nextPosition`。注意到，在图 13-2b 中，

`next`使得`nextPosition`加1，然后返回一个引用，指向的是那个位置的项Chris，这正是迭代的下一项。在图13-2c中调用`remove`删除了在`nextPosition`位置的项Chris。之后，下一项——图中的Dan——移动到线性表中下一个低位置。所以`remove`必须减小`nextPosition`的值，这样随后调用`next`时将返回Dan。



图13-2 当从线性表中删除Chris时，对线性表和`nextPosition`值的修改

上述讨论反映在`remove`的下列实现中。

```
public void remove()
{
 if (wasNextCalled)
 {
 // nextPosition was incremented by the call to next(), so
 // it is the position number of the entry to be removed
 list.remove(nextPosition);
 nextPosition--; // A subsequent call to next() must be
 // unaffected by this removal
 wasNextCalled = false; // Reset flag
 }
 else
 throw new IllegalStateException("Illegal call to remove(); "
 + "next() was not called.");
} // end remove
```



### 注：独立类迭代器

独立类迭代器必须使用ADT的公有方法访问ADT的数据。但是，某些ADT，例如栈，没有提供足够的对其数据的公有访问方法，能让这样的迭代器可行。而且，典型的独立类迭代器要比其他类型的迭代器花费更多的执行时间，因为它们不能直接访问ADT的数据。另一方面，独立类迭代器的实现通常很简单。对所给的ADT，可以同时有几个相互无关的独立类迭代器存在。

要为已经实现且其实现不会被修改的ADT提供一个迭代器，或许需要定义一个独立类迭代器。

## 内部类迭代器

13.7

通过使用独立类迭代器，可以同时对线性表进行多个不同的迭代。但是独立类迭代器属于一个公有类，所以它们仅能通过ADT操作来间接访问线性表的数据域。因此，比起其他类型的迭代器，它的迭代要花更多的时间。对于一个不是线性表的ADT来说，独立类迭代器或许没有足够的访问数据域的能力以执行迭代。

另一种选择是将迭代器类定义为 ADT 的内部类。因为得到的迭代器对象不是 ADT 的对象，所以在同一时间可以存在多个迭代过程。另外，因为迭代器属于内部类，所以它能直接访问 ADT 的数据域。由于这些原因，内部类迭代器通常比独立类迭代器更可取。

本节，我们将为 ADT 线性表的两种实现添加一个内部类，从而实现接口 `Iterator`。首先，使用线性表的链式实现，但只提供迭代器操作 `hasNext` 和 `next`。然后使用基于数组的线性表，并提供 `Iterator` 的所有三种操作。

**!** 程序设计技巧：定义内部类迭代器的类应该实现接口 `Iterable`。类的客户就能使用 `for-each` 循环来遍历类的实例中的对象了。

## 链式实现

为达到我们的目的，必须在实现 ADT 线性表的类的新内部类内定义 `Iterator` 中规范说明的方法。将这个内部类称为 `IteratorForLinkedList`，将外部类称为 `LinkedListWithIterator`。外部类非常类似于第 12 章的 `LLList`。但是它需要另一个方法，供客户用来创建迭代器。如前面的 Java 插曲所述，这个方法是 `iterator`，它声明在 `java.lang.Iterable` 标准接口中。它返回符合接口 `Iterator` 规范的一个迭代器。方法的简单实现如下所示。

```
public Iterator<T> iterator()
{
 return new IteratorForLinkedList();
} // end iterator
```

**!** 注：虽然方法 `iterator` 的这个名字符合标准，但你可能将它与接口 `Iterator` 弄混。我们想提供另外一个方法 `getIterator`，它与 `iterator` 有相同的目的。因为两个方法返回同一个迭代器，所以一个方法的实现中应该调用另一个。因为我们已经定义了 `iterator`，所以 `getIterator` 将调用 `iterator`。

为能容纳方法 `iterator` 和 `getIterator`，我们创建新的接口——如程序清单 13-2 所示——它派生于 `ListInterface` 而不是修改它。这个接口有 `ListInterface` 中的所有线性表方法及 `iterator` 和 `getIterator` 方法。

### 程序清单 13-2 接口 `ListWithIteratorInterface`

```
1 import java.util.Iterator;
2 public interface ListWithIteratorInterface<T> extends ListInterface<T>,
3 Iterable<T>
4 {
5 public Iterator<T> getIterator();
6 } // end ListWithIteratorInterface
```

在接口内显式声明方法 `iterator` 是允许的，但不是必需的，因为接口派生于 `Iterable`。因为 `Iterable` 在 `java.lang` 中，故不需要为它使用 `import` 语句。

因为类可以实现多个接口，所以我们可以在类 `LinkedListWithIterator`，让它不实现这个新接口。但实现这个接口，能让我们声明类型 `ListWithIteratorInterface` 的对象，且知道这个对象会有线性表的方法及 `iterator` 和 `getIterator` 方法。

类的框架。程序清单 13-3 概述了类 `LinkedListWithIterator` 及它的内部类 `Iterator`。

13.8

13.9

ForLinkedList 和 Node。我们将在内部类 IteratorForLinkedList 内定义接口 Iterator 中声明的方法。但是，这个迭代器不具备从数据集合中删除项的能力。

### 程序清单 13-3 类 LinkedListWithIterator 的框架

```

1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3 public class LinkedListWithIterator<T> implements ListWithIteratorInterface<T>
4 {
5 private Node firstNode;
6 private int numberofEntries;
7
8 public LinkedListWithIterator()
9 {
10 initializeDataFields();
11 } // end default constructor
12
13 <Implementations of the methods of the ADT list go here;
14 you can see them in Chapter 12, beginning at Segment 12.7>
15
16 public Iterator<T> iterator()
17 {
18 return new IteratorForLinkedList();
19 } // end iterator
20
21 public Iterator<T> getIterator()
22 {
23 return iterator();
24 } // end getIterator
25
26 <Segment 13.10 begins a description of the following inner class.>
27 private class IteratorForLinkedList implements Iterator<T>
28 {
29 private Node nextNode;
30
31 private IteratorForLinkedList()
32 {
33 nextNode = firstNode;
34 } // end default constructor
35 <Implementations of the methods in the interface Iterator go here;
36 you can see them in Segment 13.11 through 13.13.>
37
38 } // end IteratorForLinkedList
39 <Implementation of the private class Node (Listing 3-4 of Chapter 3) goes here. >
40
41 } // end LinkedListWithIterator

```

13.10

**内部类 IteratorForLinkedList。**正如在程序清单 13-3 中所见到的，私有内部类 IteratorForLinkedList 有一个数据域 nextNode，它来记录迭代过程。构造方法将这个域初始化为 firstNode，这是外部类的一个数据域，且指向含有线性表项的链表的首结点。虽然我们想象迭代器的位置位于项之间，但我们不能将它定位在结点之间。nextNode 也不能指向 next 将访问的结点之前的结点，因为首结点的前面没有结点。所以，nextNode 指向迭代中的下一个结点，即方法 next 必须访问得到的下一项的结点。



**程序设计技巧：**内部类仅通过名字就可以提及其实例的数据域，如果它没有将同样的名字用在自己的定义中。例如，内部类 IteratorForLinkedList 的构造方法使用名字可直接指向数据域 firstNode，因为不存在其他的 firstNode。不过，也可以用 LinkedListWithIterator.this.firstNode 来替代。

图 13-3 说明了一个内部类迭代器。迭代器可以直接访问 ADT 的底层数据结构——本例中是链表。因为数据域 `nextNode` 维护了迭代的当前位置，所以迭代器可以快速获取迭代中的下一项而不需要先返回到链表表头。

现在在内部类内实现接口 `Iterator` 中的方法。这些方法将是公有的，尽管它们出现在私有类中，因为它们在 `Iterator` 中是公有方法，且将被 `LinkedListWithIterator` 的客户使用。

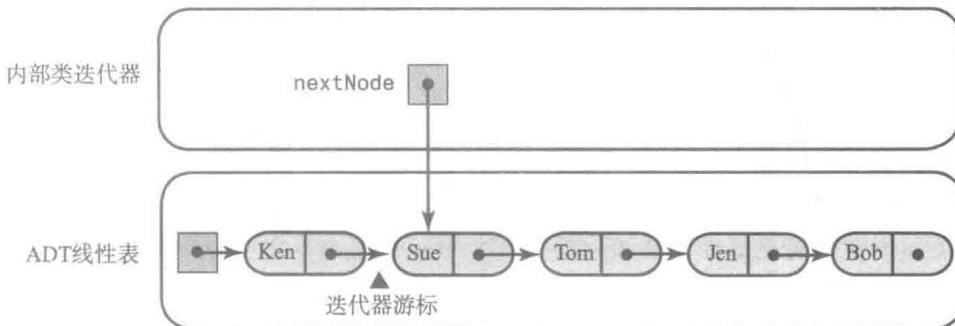


图 13-3 可直接访问实现 ADT 的链表的内部类迭代器

**方法 `next`**。如果迭代没到达最后位置，则 `nextNode` 指向含有迭代下一项的结点。所以 `next` 能够很容易地得到指向这个项的引用。然后方法必须让 `nextNode` 前进到下一个结点，并返回获得的线性表项。但是，如果迭代已经到达最后位置，则 `next` 方法必须抛出一个异常。

```
public T next()
{
 T result;
 if (hasNext())
 {
 result = nextNode.getData();
 nextNode = nextNode.getNextNode(); // Advance iterator
 }
 else
 throw new NoSuchElementException("Illegal call to next(); " +
 "iterator is after end of list.");
 return result; // Return next entry in iteration
} // end next
```

**方法 `hasNext`**。方法 `next` 返回迭代最后一项后，`nextNode` 将是 `null`，因为 `null` 是链表中最后结点的链接部分。方法 `hasNext` 可以简单地比较 `nextNode` 与 `null`，来看看迭代是否到达最后：

```
public boolean hasNext()
{
 return nextNode != null;
} // end hasNext
```



学习问题 3 当线性表为空时，方法 `hasNext` 返回什么？为什么？

**方法 `remove`**。即使我们决定不为这个迭代器提供 `remove` 操作，但也必须实现这个方法，因为它声明在接口 `Iterator` 中。如果客户调用 `remove`，方法只是简单地抛出一个运行时异常 `UnsupportedOperationException`。下面给出 `remove` 定义的一个示例。

13.11

13.12

13.13

```

public void remove()
{
 throw new UnsupportedOperationException("remove() is not supported " +
 "by this iterator");
} // end remove

```

这个异常在包 `java.lang` 中，所以自动含在每个 Java 程序中。故不需要使用 `import` 语句。



### 注：remove 方法

遍历过程中不允许删除项的迭代器并不罕见。这种情形下，可以定义 `remove` 方法，但如果调用，它抛出一个异常。



### 注：内部类迭代器

内部类迭代器可以直接访问 ADT 的数据，所以一般地它可能比独立类迭代器运行得更快。但它的实现通常更麻烦一些。这两个迭代器都还有另一个优点：同一时刻可以有几个迭代器对象并存，且各自独立地遍历线性表。



**学习问题 4** 给出类 `LinkedListWithIterator`，创建 Java 插曲段 J4.11 中提到的迭代器 `nameIterator` 和 `countingIterator` 的 Java 语句分别是什么？

**学习问题 5** 修改第 10 章程序清单 10-2 中显示的方法 `displayList`，使之能用在类 `LinkedListWithIterator` 的客户中，使用迭代器方法显示线性表。

## 基于数组的实现

13.14

对于基于数组的实现，我们的迭代器将提供 `remove` 方法。我们从第 11 章给出的 ADT 线性表基于数组的实现 `AList` 入手。程序清单 13-4 中显示的新类，与类 `AList` 有同样的数据域和方法。但因为新类实现了接口 `ListWithIteratorInterface`，所以它还包含了方法 `iterator` 和 `getIterator`。我们的类还包含内部类 `IteratorForArrayList`，它实现了接口 `Iterator`。

**程序清单 13-4** `ArrayListWithIterator` 类的框架

```

1 import java.util.Arrays;
2 import java.util.Iterator;
3 import java.util.NoSuchElementException;
4 public class ArrayListWithIterator<T> implements ListWithIteratorInterface<T>
5 {
6 private T[] list; // Array of list entries; ignore list[0]
7 private int numberOfEntries;
8 private boolean integrityOK;
9 private static final int DEFAULT_CAPACITY = 25;
10 private static final int MAX_CAPACITY = 10000;
11
12 public ArrayListWithIterator()
13 {
14 this(DEFAULT_CAPACITY);
15 } // end default constructor
16
17 public ArrayListWithIterator(int initialCapacity)
18 {
19 integrityOK = false;
20

```

```

21 // Is initialCapacity too small?
22 if (initialCapacity < DEFAULT_CAPACITY)
23 initialCapacity = DEFAULT_CAPACITY;
24 else // Is initialCapacity too big?
25 checkCapacity(initialCapacity);
26
27 // The cast is safe because the new array contains null entries
28 @SuppressWarnings("unchecked")
29 T[] tempList = (T[])new Object[initialCapacity + 1];
30 list = tempList;
31 numberEntries = 0;
32 integrityOK = true;
33 } // end constructor
34
35 < Implementations of the methods of the ADT list go here ;
36 you can see them in Chapter 11, beginning at Segment 11.5. >
37
38 public Iterator<T> iterator()
39 {
40 return new IteratorForArrayList();
41 } // end iterator
42
43 public Iterator<T> getIterator()
44 {
45 return iterator();
46 } // end getIterator
47
48 < Segment 13.15 begins a description of the following inner class. >
49 private class IteratorForArrayList implements Iterator<T>
50 {
51 private int nextIndex; // Index of next entry
52 private boolean wasNextCalled; // Needed by remove
53
54 private IteratorForArrayList()
55 {
56 nextIndex = 1; // Begin at list's first entry
57 wasNextCalled = false;
58 } // end default constructor
59
60 < Implementations of the methods in the interface Iterator go here;
61 you can see them in Segments 13.16 through 13.18. >
62
63 } // end IteratorForArrayList
64 } // end ArrayListWithIterator

```

**内部类 IteratorForArrayList。**正如你可以用手指跟踪在本页的位置一样，迭代器可以使用下标来跟踪迭代器在线性表项所用数组内的位置。称为 `nextIndex` 的这个下标，是私有内部类 `IteratorForArrayList` 的数据域。它将是迭代中下一项的下标。构造方法将 `nextIndex` 初始化为 1，如程序清单 13-4 所示，因为在我们对 ADT 线性表的实现中，线性表项占据数组中下标从 1 开始的位置。

正如之前在段 13.3 和段 13.6 中所述，要为迭代器提供删除操作，就必须提供一个额外的数据域，可供 `remove` 方法查看 `next` 是否已被调用。重申一下，我们将这个数据域称为 `wasNextCalled`，不过这里，它定义在内部类中。构造方法将这个域初始化为假。

**方法 `hasNext`。**如果 `nextIndex` 小于等于线性表长，则迭代器能获取下一项。所以，`hasNext` 有下列简单的实现：

```

public boolean hasNext()
{
 return nextIndex <= numberEntries;
}

```

13.15

13.16

```
 } // end hasNext
```

注意到，当线性表为空时，即当 `numberOfEntries` 为 0 时，`hasNext` 返回假。

13.17

**方法 next。**方法 `next` 的实现与段 13.5 中给出的独立类迭代器的版本有相同的一般形式。如果 `hasNext` 返回真，则 `next` 返回迭代的下一项。这里，下一项是 `list[nextIndex]`。方法增大 `nextIndex` 的值而使迭代前进，而且将标识 `wasNextCalled` 设置为真。另一方面，如果 `hasNext` 返回假，则 `next` 方法抛出一个异常。

```
public T next()
{
 checkIntegrity();
 if (hasNext())
 {
 wasNextCalled = true;
 T nextEntry = list[nextIndex];
 nextIndex++; // Advance iterator
 return nextEntry;
 }
 else
 throw new NoSuchElementException("Illegal call to next(); " +
 "iterator is after end of list.");
} // end next
```

13.18

**方法 remove。**从线性表中删除一项，涉及数组 `list` 中的项的移动。因为我们已经开发了线性表的 `remove` 方法的代码，所以可以调用它，而不是直接访问数组 `list`。为此，需要知道被删除的线性表项的下标位置。回忆段 10.1，线性表中项的下标位置从 1 开始，所以这与本实现中的数组下标是一致的。

图 13-4 说明了在本实现中如何使用 `nextIndex`。图中显示了调用 `next` 之前、调用 `next` 之后但调用 `remove` 之前、调用 `remove` 之后的线性表项所在的数组和下标 `nextIndex`。图 13-4b 显示 `next` 返回指向迭代中下一项 Chris 的引用，然后 `nextIndex` 加 1。方法 `remove` 必须从线性表中删除这个项。但现在 `nextIndex` 比 Chris 的下标大 1。即 `nextIndex` 比必须删除的线性表项的位置数大 1。所以 `remove` 操作让 `nextIndex` 的值减 1，`nextIndex` 现在是 Deb——删除 Chris 后迭代中的下一项——的下标，如图 13-4c 所示。

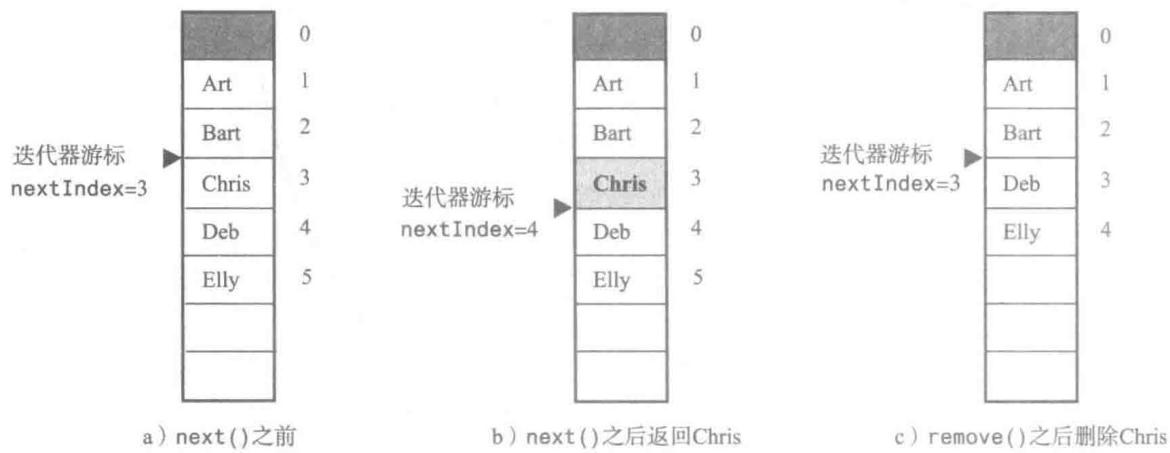


图 13-4 当从线性表中删除 Chris 时线性表项的数组及 `nextIndex` 的改变

内部类 `IteratorForArrayList` 中的方法 `remove` 有下列定义：

```
public void remove()
```

```

{
 checkIntegrity();
 if (wasNextCalled)
 {
 // nextIndex was incremented by the call to next, so it is
 // 1 larger than the position number of the entry to be removed
 ArrayListWithIterator.this.remove(nextIndex - 1);
 nextIndex--; // Index of next entry in iteration
 wasNextCalled = false; // Reset flag
 }
 else
 throw new IllegalStateException("Illegal call to remove(); " +
 "next() was not called.");
} // end remove

```

要在迭代器的 `remove` 方法内，调用定义在外部类内的线性表的 `remove` 方法，必须在线性表的方法名之前加上 `ArrayListWithIterator.this`。



**学习问题 6** 考虑图 13-4 中的线性表及对 `next` 和 `remove` 的调用。

- 在图 13-4c 中如果调用 `remove` 之后调用 `next`，则返回什么？
- 在图 13-4b 中如果调用 `next` 之后调用 `next`，则返回什么？

**学习问题 7** 如果构造方法将 `nextIndex` 设置为 0 而不是 1，则内部类 `IteratorForArrayList` 中的方法应做哪些修改？

## 为什么迭代器方法在它自己的类中

独立类迭代器和内部类迭代器都能同时对数据集有多个不同的迭代。因为内部类迭代器可以直接访问含有 ADT 数据的结构，故它们能比独立类迭代器运行得更快，所以通常更可取。

为什么我们不简单地将迭代器操作看作 ADT 附加的操作呢？为回答这个问题，我们修改第 12 章给出的线性表的链式实现，让它包含 Java 接口 `Iterator` 中规范说明的方法。为让这个实现简单，我们没有提供 `Iterator` 中规范说明的删除操作。得到的类概述在程序清单 13-5 中，实际上与段 13.9 所描述的实现了一个内部类迭代器的类 `LinkedListWithIterator` 非常相似。两个类的不同之处已标记出来。

段 13.9 中所展示的内部类 `IteratorForLinkedList` 没有出现在新类中，但其数据域 `nextNode` 和其迭代器方法 `hasNext`、`next` 和 `remove` 并未改变。我们用公有方法 `resetTraversal` 替代内部类的构造方法将 `nextNode` 设置为 `firstNode`。在遍历前将调用这个方法。

**程序清单 13-5** 类 `ListWithTraversal` 的框架

```

1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3 public class ListWithTraversal<T> implements ListInterface<T>, Iterator<T>
4 {
5 private Node firstNode;
6 private int numberOfEntries;
7 private Node nextNode; // Node containing next entry in iteration
8
9 public ListWithTraversal()
10 {
11 initializeDataFields();
12 } // end default constructor

```

```

13
14 < Implementations of the remaining methods of the ADT list go here ;
15 you can see them in Chapter 12, beginning at Segment 12.7>
16 .
17 private void initializeDataFields()
18 {
19 firstNode = null;
20 numberofEntries = 0;
21 nextNode = null;
22 } // end initializeDataFields
23
24 < Implementations of the methods in the interface Iterator go here;
25 you can see them in Segments 13.11 through 13.13.>
26 .
27 public void resetTraversal()
28 {
29 nextNode = firstNode;
30 } // end resetTraversal
31
32 < Implementation of the private class Node (Listing 3-4 of Chapter 3) goes here.>
33 } // end ListWithTraversal

```

**13.20** **示例：遍历线性表。**如果 myList 是前一个类 ListWithTraversal 的实例，则它有 ADT 线性表的方法及 Iterator 中的方法。所以，如果使用像 myList.add("Chris") 这样的调用，将字符串添加到 myList 中，则可以如下显示线性表：

```

myList.resetTraversal();
while (myList.hasNext())
 System.out.println(myList.next());

```

要设置从头遍历线性表，则必须调用 resetTraversal。注意到，可以使用线性表 myList，而不是用迭代器对象来调用 Iterator 的方法。段 13.2 中的处理正好相反。



**学习问题 8** 修改第 10 章程序清单 10-2 中所示的用于类 ListWithTraversal 的客户的方法 displayList，使用前一个示例中的方法显示线性表。这个实现有不足之处吗？解释之。

**学习问题 9** 假定你想去掉方法 resetTraversal。

- 默认构造方法能将 nextNode 初始化为 firstNode 吗？解释之。
- add 方法能将 nextNode 初始化为 firstNode 吗？解释之。

**13.21**

**这个方法有什么问题？**虽然这些遍历方法都可以快速执行，因为它们可以直接访问线性表的底层数据结构，但将它们作为线性表操作是有坏处的。同一时间只能有一个遍历在执行。另外，像 resetTraversal 这样的操作，它不属于接口 Iterator，却是初始化遍历时所必需的。结果就是，得到的 ADT 有太多的方法；它承受着接口膨胀 (interface bloat) 之苦。更重要的是，线性表的功能与迭代器的功能混为一谈，这是不佳的设计方案。从概念上讲，当你指向书页上的一行行字时，你的手指不是书的一部分。它是用来跟踪书页上字符的独立对象。

编程时多做点额外的努力，就可以将迭代器方法组织为内部类。这样做，既得到快的运行速度也没什么不利因素。

## 基于数组实现接口 ListIterator

**13.22**

与本章之前对接口 Iterator 的处理一样，让实现接口 ListIterator 的类，作为使

用数组表示 ADT 线性表的类的内部类。我们先在程序清单 13-6 中定义接口，隐式声明了 ADT 线性表的操作，显式声明了方法 `getIterator`。这种情形下，`getIterator` 的返回类型是 `ListIterator<T>`，而不是 `Iterator<T>`。因为我们的接口还派生于 `Iterable`，所以它隐式声明了方法 `iterator`，其返回类型是 `Iterator<T>`。

### 程序清单 13-6 接口 `ListWithListIteratorInterface`

```

1 import java.util.ListIterator;
2 public interface ListWithListIteratorInterface<T> extends Iterable<T>,
3 ListInterface<T>
4 {
5 public ListIterator<T> getIterator();
6 } // end ListWithListIteratorInterface

```

实现 ADT 线性表的类。我们的类与第 11 章给出的类 `AList` 有同样的数据域和方法，包括方法 `iterator` 和 `getIterator`。类还包含内部类 `ListIteratorForArrayList`，它实现了接口 `ListIterator`。程序清单 13-7 显示了新线性表类的形式。13.23

### 程序清单 13-7 类 `ArrayListWithListIterator` 的框架

```

1 import java.util.Arrays;
2 import java.util.Iterator;
3 import java.util.ListIterator;
4 import java.util.NoSuchElementException;
5 public class ArrayListWithListIterator<T>
6 implements ListWithListIteratorInterface<T>
7 {
8 private T[] list; // Array of list entries; ignore list[0]
9 private int numberOfEntries;
10 private boolean integrityOK;
11 private static final int DEFAULT_CAPACITY = 25;
12 private static final int MAX_CAPACITY = 10000;
13
14 public ArrayListWithListIterator()
15 {
16 this(DEFAULT_CAPACITY);
17 } // end default constructor
18
19 public ArrayListWithListIterator(int initialCapacity)
20 {
21 integrityOK = false;
22
23 // Is initialCapacity too small?
24 if (initialCapacity < DEFAULT_CAPACITY)
25 initialCapacity = DEFAULT_CAPACITY;
26 else // Is initialCapacity too big?
27 checkCapacity(initialCapacity);
28
29 // The cast is safe because the new array contains null entries
30 @SuppressWarnings("unchecked")
31 T[] tempList = (T[])new Object[initialCapacity + 1];
32 list = tempList;
33 numberOfEntries = 0;
34 integrityOK = true;
35 } // end constructor
36
37 < Implementations of the methods of the ADT list go here;
38 you can see them in Chapter 11, beginning at Segment 11.5. >
39
40 public ListIterator<T> getIterator()
41 {

```

```

42 return new ListIteratorForArrayList();
43 } // end getIterator
44
45 public Iterator<T> iterator()
46 {
47 return getIterator();
48 } // end iterator
49 .
50 private class ListIteratorForArrayList implements ListIterator<T>
51 {
52 <The description of this implementation begins with Segment 13.24.>
53 .
54 } // end ListIteratorForArrayList
55 } // end ArrayListWithListIterator

```

## 内部类

**13.24** 数据域和构造方法。我们从考虑方法 `remove` 和 `set` 如何抛出异常 `IllegalStateException` 入手，着手实现内部类 `ListIteratorForArrayList`。这两个方法因为同样的原因抛出这个异常，即如果以下两条至少成立一条：

- 没有调用 `next` 或 `previous`，或者
- 从上次调用 `next` 或 `previous` 后，`remove` 或 `add` 已被调用

图 13-5 显示了能引发 `IllegalStateException` 异常的 `remove` 的不同调用情况。

|                                      |                                                      |
|--------------------------------------|------------------------------------------------------|
| a) <code>traverse.remove();</code>   | 导致异常； <code>next</code> 和 <code>previous</code> 都没调用 |
| b) <code>traverse.next();</code>     |                                                      |
| <code>traverse.remove();</code>      | 合法                                                   |
| <code>traverse.remove();</code>      | 导致异常                                                 |
| c) <code>traverse.previous();</code> |                                                      |
| <code>traverse.remove();</code>      | 合法                                                   |
| <code>traverse.remove();</code>      | 导致异常                                                 |
| d) <code>traverse.next();</code>     |                                                      |
| <code>traverse.add(...);</code>      |                                                      |
| <code>traverse.remove();</code>      | 导致异常                                                 |
| e) <code>traverse.previous();</code> |                                                      |
| <code>traverse.add(...);</code>      |                                                      |
| <code>traverse.remove();</code>      | 导致异常                                                 |

图 13-5 调用时迭代器 `traverse` 的 `remove` 方法抛出异常的几种情况

这样的实现最初可能令人生畏，但并非一定这么困难。当实现段 13.18 中 `Iterator` 的 `remove` 方法时，我们检测布尔数据域 `wasNextCalled`，看看 `next` 是否被调用过。在这里也可以这样处理，为方法 `previous` 和 `add` 定义类似的域，但从逻辑上来看用不了这么多。相反，定义一个布尔域来检查 `remove` 或 `set` 的调用是否合法：

```
private boolean isRemoveOrSetLegal;
```

如果 `remove` 或 `set` 发现这个数据域的值为假，则将抛出 `IllegalStateException`。

这个数据域应该由构造方法初始化为假。方法 `next` 和 `previous` 应该将它设置为真，而方法 `add` 和 `remove` 应该将它设置为假。

`remove` 和 `set` 必须知道调用的是 `next` 还是 `previous`，这样它们才能正确访问线性表项。所以，我们定义一个数据域来记录对这些方法的最后一次调用，且用一个枚举类型量来保存这个域的值。下列枚举定义就足够了：

```
private enum Move {NEXT, PREVIOUS}
```

因为枚举确实是一个类，所以我们将它定义在内部类 `ListIteratorForArrayList` 的外面，但在 `ArrayListWithListIterator` 的内部。该数据域的定义很简单：

```
private Move lastMove;
```

除了这两个数据域，还需要一个域 `nextIndex` 来记录迭代中下一项的下标。这个域与我们之前在段 13.15 中描述的一样。所以内部类的开头如下所示：

```
private class ListIteratorForArrayList implements ListIterator<T>
{
 private int nextIndex; // Index of next entry in the iteration
 private boolean isRemoveOrSetLegal;
 private Move lastMove;

 private ListIteratorForArrayList()
 {
 nextIndex = 1; // Iteration begins at list's first entry
 isRemoveOrSetLegal = false;
 lastMove = null;
 } // end default constructor
 ...
}
```

方法 `hasNext`。`hasNext` 与之前在段 13.16 中的实现一样。回忆一下，如果迭代器没有到达线性表尾，则它返回真。13.25

```
public boolean hasNext()
{
 return nextIndex <= numberofEntries;
} // end hasNext
```

方法 `next`。`next` 的实现与段 13.17 中给出的类似。但此处，它要设置不同的数据域。将 `lastMove` 设置为 `Move.NEXT`，将布尔域 `isRemoveOrSetLegal` 设置为真。13.26

```
public T next()
{
 if (hasNext())
 {
 lastMove = Move.NEXT;
 isRemoveOrSetLegal = true;
 T nextEntry = list[nextIndex];
 nextIndex++; // Advance iterator
 return nextEntry;
 }
 else
 throw new NoSuchElementException("Illegal call to next(); " +
 "iterator is after end of list.");
} // end next
```

方法 `hasPrevious` 和 `previous`。`hasPrevious` 和 `previous` 的实现分别类似于 `hasNext` 和 `next` 的实现。13.27

```
public boolean hasPrevious()
```

```

 {
 return (nextIndex > 1) && (nextIndex <= numberofEntries + 1);
 } // end hasPrevious

 public T previous()
 {
 if (hasPrevious())
 {
 lastMove = Move.PREVIOUS;
 isRemoveOrSetLegal = true;
 T previousEntry = list[nextIndex - 1];
 nextIndex--; // Move iterator back
 return previousEntry;
 }
 else
 throw new NoSuchElementException("Illegal call to previous(); " +
 "iterator is before beginning of list.");
 } // end previous
}

```

**13.28 方法 nextIndex 和 previousIndex。**方法 nextIndex 在调用 next 的情况下，返回 next 方法将返回的项的下标，或者当迭代器到达线性表尾之后返回线性表的大小。虽然变量 nextIndex 从 1 开始计数，但记住，ListIterator 对象从 0 开始。

```

public int nextIndex()
{
 int result;
 if (hasNext())
 result = nextIndex - 1; // Change to zero-based numbering of iterator
 else
 result = numberofEntries; // End-of-list flag
 return result;
} // end nextIndex

```

方法 previousIndex 在调用 previous 的情况下，返回 previous 方法将返回项的下标，或是当迭代器位于线性表头之前时返回 -1。

```

public int previousIndex()
{
 int result;
 if (hasPrevious())
 result = nextIndex - 2; // Change to zero-based numbering of iterator
 else
 result = -1; // Beginning-of-list flag
 return result;
} // end previousIndex

```

**13.29 方法 add。**方法 add 将项插入线性表中迭代器当前位置之前的地方，即在项 list[nextIndex] 之前，如图 13-6 所示。为避免代码重复，我们调用线性表的 add 方法将项添加在线性表内的 nextIndex 位置处。回忆一下，新项之后的项都要移动并重新编号。所以，我们需要将 nextIndex 加 1，以便它能继续标记后续调用 next 时要返回的项。故 add 的实现如下所示。

```

public void add(T newEntry)
{
 isRemoveOrSetLegal = false;
 // Insert newEntry immediately before the the iterator's current position
 ArrayListWithListIterator.this.add(nextIndex, newEntry);
 nextIndex++;
} // end add

```

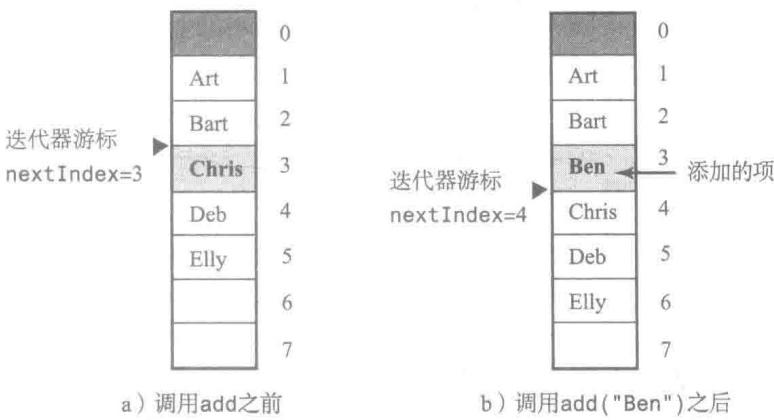


图 13-6 将 Ben 添加入线性表时改变了线性表项所在的数组及 nextIndex

**方法 remove。**如果在调用 `remove` 方法之前调用了 `next`, 则 `remove` 的逻辑类似于段 13.18 中见过的接口 `Iterator` 中 `remove` 方法的逻辑。回忆图 13-4 中说明的, 在调用 `next` 和 `remove` 之前及之后, 线性表项的数组及下标 `nextIndex`。图 13-7 给出了类似的图示, 显示在调用 `remove` 之前调用 `previous` 时的情形。在图 13-7b 中, `previous` 返回的引用指向的是迭代中的前一项——Bart, 且 `nextIndex` 减 1。方法 `remove` 必须从线性表中删除这个项。在图 13-7c 中删除项 Bart 后, 下一项——Chris——移到数组的前一个位置。所以 `nextIndex` 仍是迭代中下一项的下标, 故不变。

13.30



图 13-7 从线性表中删除 Chris 时改变了线性表项的数组及 nextIndex 值

记住, 如果数据域 `isRemoveOrSetLegal` 为假, 则 `remove` 必须抛出异常。如果数据域为真, 则方法必须将它设置为假。`remove` 的实现如下所示。

```
public void remove()
{
 if (isRemoveOrSetLegal)
 {
 isRemoveOrSetLegal = false;
 if (lastMove.equals(Move.NEXT))
 {
 // next() called, but neither add() nor remove() has been
 // called since
 // Remove entry last returned by next()
 // nextIndex is 1 more than the index of the entry returned
 // by next()
 }
 }
}
```

```

 ArrayListWithListIterator.this.remove(nextIndex - 1);
 nextIndex--; // Move iterator back
 }
 else
 {
 // previous() called, but neither add() nor remove() has been
 // called since
 // Remove entry last returned by previous()
 // nextIndex is the index of the entry returned by previous()
 ArrayListWithListIterator.this.remove(nextIndex);
 } // end if
}
else
 throw new IllegalStateException("Illegal call to remove(); " +
 "next() or previous() was not called, OR"
 "'add()' or 'remove()' called since then.");
} // end remove

```

13.31

**方法 set。**方法 set 替换最后一次调用 next 或 previous 时返回的线性表项。它使用方法 next 或 previous 更新后的 nextIndex。因为方法 next 返回位于 nextIndex 的线性表项，然后 nextIndex 的值增 1，所以接下来调用方法 set 时应该替换的是位置 nextIndex-1 处的项。类似地，因为 previous 返回位于 nextIndex-1 处的线性表项，然后 nextIndex 的值减 1，所以调用 previous 后，方法 set 应该替换位置 nextIndex 处的项。

这些结论反映在 set 的下列实现中，且是否抛出 IllegalStateException 的逻辑与 remove 中使用的一样。

```

public void set(T newEntry)
{
 if (isRemoveOrSetLegal)
 {
 if (lastMove.equals(Move.NEXT))
 list[nextIndex - 1] = newEntry; // Replace entry last returned by
 // next()
 else
 {
 // Assertion: lastMove.equals(Move.PREVIOUS)
 list[nextIndex] = newEntry; // Replace entry last returned by
 // previous()
 } // end if
 }
 else
 throw new IllegalStateException("Illegal call to set(); " +
 "next() or previous() was not called, OR"
 "'add()' or 'remove()' called since then.");
} // end set

```



**设计决策：**当定义方法 set 时，我们是要清晰还是要效率？

清晰与效率是程序设计时常常要权衡的问题。在前面的 set 方法中遇到过这个情况。如果我们调用 ADT 线性表的方法 replace 替代给数组元素赋值，则方法的定义可能更清楚，即更易于理解。但是，这样做会像独立类迭代器一样降低效率。因为我们定义的是一个内部类迭代器，故时间效率是我们的目标。



**注：**当相关的 ADT 基于数组实现而不是链式实现时，将整个接口 ListIterator 实现为一个内部类更容易些。(见练习 17。)

 注：当类型 `ListIterator` 的迭代器不提供 `add`、`remove` 和 `set` 操作时，它的实现更简单。这样一个迭代器是有用的，因为它能让你双向遍历线性表。我们将它的实现留作练习。

## 本章小结

- 接口 `Iterator` 规范说明了 3 个方法：`hasNext`、`next` 和 `remove`。实现这个接口的迭代器不需要提供删除操作。相反，方法 `remove` 应该抛出异常 `UnsupportedOperationException`。
- 接口 `ListIterator` 中规范说明了 9 个方法，包括 `Iterator` 中规范说明的 3 个方法。它们是 `hasNext`、`next`、`hasPrevious`、`previous`、`nextIndex`、`previousIndex`、`add`、`remove` 和 `set`。这里方法 `add`、`remove` 和 `set` 是可选的，它们可以抛出异常 `UnsupportedOperationException` 而不是对线性表施加操作。
- 可以将接口 `Iterator` 和 `ListIterator` 实现为自己的类。这个类可以是所讨论的实现 ADT 的类的内部类，或是它可以是公有的且独立于 ADT 的类。
- 内部类迭代器能有多个独立的遍历集合的迭代器。它还允许迭代器直接访问底层数据结构，所以它的实现效率高。
- 独立类迭代器也能允许同时存在多个不同的迭代。但是，因为迭代器仅通过 ADT 操作间接访问线性表的数据域，所以迭代时比内部类迭代器要花更多的时间。另一方面，它的实现常常更简单。
- 某些 ADT 没有提供足够的对其数据的公有访问方法，这就没有可能使用独立类迭代器。但是，要为已经实现且其实现不会被修改的 ADT 提供一个迭代器，或许需要定义一个独立类迭代器。

## 程序设计技巧

- 定义内部类迭代器的类，应该实现接口 `Iterable`。这样，类的客户可以使用 `foreach` 循环来遍历类的实例中的对象。

## 练习

1. 假定 `nameList` 是含有下列字符串的线性表：Kyle, Cathy, Sam, Austin, Sara。执行下列语句序列得到的输出是什么？

```
Iterator<String> nameIterator = nameList.getIterator();
System.out.println(nameIterator.next());
nameIterator.next();
System.out.println(nameIterator.next());
nameIterator.remove();
System.out.println(nameIterator.next());
displayList(nameList);
```

2. 重做练习 1，但换成下列语句：

```
Iterator<String> nameIterator = nameList.getIterator();
nameIterator.next();
nameIterator.remove();
nameIterator.next();
nameIterator.next();
nameIterator.remove();
```

```

System.out.println(nameIterator.next());
displayList(nameList);
System.out.println(nameIterator.next());
System.out.println(nameIterator.next());

```

3. 假定 nameList 是至少含有一个字符串的线性表，且 nameIterator 定义如下：

```
Iterator<String> nameIterator = nameList.getIterator();
```

写出使用 nameIterator 显示线性表中最后一个字符串的 Java 语句。

4. 给定练习 3 描述的 nameList 和 nameIterator，写出使用 nameIterator 从后向前显示线性表中所有字符串的语句。
5. 给定练习 3 描述的 nameList 和 nameIterator，写出使用 nameIterator 删除线性表中所有项的语句。
6. 给定练习 3 描述的 nameList 和 nameIterator，写出使用 nameIterator 从线性表中删除字符串 CANCEL 的所有出现的语句。
7. 给定练习 3 描述的 nameList 和 nameIterator，写出使用 nameIterator 从线性表中删除任意重复项的语句。
8. 给定练习 3 描述的 nameList 和 nameIterator，写语句，使用 nameIterator 统计线性表中每个字符串出现的次数，不允许改变线性表且不能重复计算。
9. 假定 aList 和 bList 是 java.util.ArrayList 的实例。使用两个迭代器，找到并显示两个线性表中共有的所有对象。不能改变任何一个线性表的内容。
10. 假定 aList 和 bList 是 java.util.ArrayList 的实例，其按从小到大的次序含有 Comparable 对象。使用两个迭代器，将对象从 bList 中移到 aList 中合适的位置。当处理完毕，aList 中的对象应该有序，而 bList 应该为空。
11. 修改段 13.3 中概述的类 SeparateIterator，让它不提供删除操作。尽可能简化这个类。
12. 假定一个类实现了段 13.8 程序清单 13-2 中所给的接口 ListWithIteratorInterface。假定 aList 是这个类的一个实例，且无序含有 Comparable 对象。使用一个迭代器，在类内实现下列两个方法：
  - a. getMin 返回线性表中最小的对象
  - b. removeMin 删除并返回线性表中最小的对象
13. 重做前一个练习，但使用 for-each 循环替代迭代器。
14. 假定 nameList 是线性表，含有下列字符串：Kyle、Cathy、Sam、Austin 和 Sara。执行下列语句得到的输出是什么？

```

ListIterator<String> nameIterator = nameList.getIterator();
System.out.println(nameIterator.next());
nameIterator.next();
nameIterator.next();
System.out.println(nameIterator.next());
nameIterator.set("Brittany");
nameIterator.previous();
nameIterator.remove();
System.out.println(nameIterator.next());
displayList(nameList);

```

15. 使用下列语句重做前一个练习：

```

ListIterator<String> nameIterator = nameList.getIterator();
nameIterator.next();
nameIterator.remove();
nameIterator.next();
nameIterator.next();
nameIterator.previous();

```

```

nameIterator.remove();
System.out.println(nameIterator.next());
nameIterator.next();
nameIterator.set("Brittany");
System.out.println("Revised list:");
displayList(nameList);
System.out.println(nameIterator.previous());
System.out.println(nameIterator.next());

```

16. 给定字符串线性表及迭代器 nameIterator，其数据类型是 `ListIterator`，写出语句，将字符串 Bob 添加在字符串 Sam 的第一次出现之后。

17. 如果你想使用链式方式将接口 `ListIterator` 实现为内部类迭代器，要面对的困难有哪些？

## 项目

- 修改段 13.9 描述的类 `LinkedListWithIterator`。让内部类 `IteratorForLinkedList` 提供删除操作。你需要另一个内部类中的数据域 `priorNode`，指向下一个结点前的结点。
- 实现作为独立类迭代器的接口 `java.util.ListIterator` 中的所有方法。
- 将接口 `java.util.ListIterator` 实现为类 `LList` (第 12 章给出) 的内部类，但不提供 `add`、`remove` 和 `set` 方法。
- 实现含有删除操作的迭代器，作为采用下列方式实现的包类的内部类
  - 基于数组
  - 链式
- 使用内部类为栈类实现迭代器，栈类的实现方式是
  - 基于数组
  - 链式
 迭代器应该提供删除操作吗？
- 有时对一组数据执行的一个统计操作是，删除远离平均值的值。写一个程序，从文本文件读入实数，每行一个。将数据值作为 `Double` 对象保存在类 `java.util.ArrayList` 的实例中。然后
  - 使用迭代器计算值的平均值及标准差。显示这些结果。
  - 使用第二个迭代器删除与平均值偏差两倍标准差的值。
  - 使用 `for-each` 循环显示剩余的值，并计算新的平均值。显示新的平均值。
 如果数据值是  $x_1, x_2, \dots, x_n$ ，则它们的平均值  $\mu$  是它们的和除以  $n$ ，它们的标准差是

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$$

7. 考虑下列情形。你创建一个线性表，然后在其中添加 10 项。得到线性表的一个迭代器并两次调用 `next` 前移。使用线性表的 `remove` 方法从线性表中删除前 5 项。然后调用迭代器的 `remove` 方法，期望删除方法 `next` 最后返回的项。但是，这个项已经从表中删除了。使用迭代器改变线性表的状态，正如你这里所做的，可能导致迭代器无法预料的事情发生。

修改接口 `Iterator` 中的方法，如果在迭代器创建后且方法被调用前线性表的状态被改变了，让方法抛出异常 `StateChangedException`。对应于 `Iterator` 的修改，修改项目 1 描述的 `LinkedListWithIterator` 的实现。

- 修改段 13.14 中概述的类 `ArrayListWithIterator`，它派生于第 11 章讨论的类 `AList`。
- 重做第 10 章的项目 9 和项目 10，为 ADT 食谱增加配料表和操作指南的迭代器。
- (游戏) 考虑纸牌匹配游戏，你有 10 ~ 99 之间的随机整数列表。若列表中相邻的一对整数中，其第一位或第二位相等，则从列表中删除它们。如果所有的值都删除了，则你赢了。  
例如，考虑下列 10 个整数的序列：

10 82 43 23 89 12 43 84 23 32

整数对 10 和 82 在哪个数位都不等，所以不能被删除。但是，整数对 43 和 23 在第二个数位等，可以删除，余下如下的序列：

10 82 89 12 43 84 23 32

从删除数对之后的值 89 开始继续检查整数对。没有能配对的其他整数。现在返回到序列开头，检查数对。整数对 82 和 89 在第一个数位等，可以删除。

10 12 43 84 23 32

不能再删除其他数对了，所以我们输了。

写一个模拟这个游戏的程序。它应该生成 40 个随机的两位数整数，并将它们放到 `java.util.ArrayList` 的实例中，使用 `ListIterator` 的实例。然后，使用这个迭代器，扫描列表并删除匹配的整数对。删除每对数后，使用迭代器显示表中剩余的值。

- 11.(游戏) 假定线性表中含有 1000 个项，其值是随机的，介于 2 ~ 12 之间，包括 2 和 12。这些值表示一对骰子数。开始时迭代器位于线性表位置 1。令  $d$  是位置 1 处的项值，将项设置为 0 且让迭代器前移  $d$  个位置。在这个新位置，重复过程：获取项值，将项设置为 0，移动迭代器。继续重复这个过程，直到迭代器到达线性表尾时为止。

当迭代器到达线性表表尾时，反转方向，让迭代器朝着线性表位置 1 的方向移动。并重复刚才的步骤。如果迭代器检测到含有 0 的项，则游戏应该反复“掷骰子”，即生成一个随机数，直到得到 7 时为止。此时，再次掷骰子，迭代器移动相当的位置数。游戏中，记录掷骰子的总次数及迭代器移动的总次数。继续游戏，直到迭代器遍历了线性表 4 次并回到位置 1 时为止。掷骰子数加上迭代器的移动数，将和值作为游戏的得分。

设计并实现这个游戏。创建游戏的几个实例，看看哪个的得分最低。

# 使用递归求解问题

先修章节：第 5 章、第 9 章

## 目标

学习完本章后，应该能够

- 求解汉诺塔问题
- 认识到什么时候不应该使用递归
- 描述给定语法的语言中的字符串
- 为给定语言写语法
- 认识间接递归
- 使用递归和回溯求解问题

第 9 章介绍了递归。我们编写返回一个值的递归方法和递归的 void 方法。我们的例子处理数组和链表。那一章还介绍了如何评估递归方法的时间效率。本章，我们将使用递归求解不涉及数学或数据结构知识的问题。

## 困难问题的简单求解方案

汉诺塔是计算机科学的一个经典问题，它的求解不是显而易见的。假定有 3 根柱子及多个不同直径的圆盘。每个盘子的中间有一个孔，这样它能插到每根柱子上。假定盘子已经按从最大到最小的次序放到第一根柱子上，最小的盘子在最上面。14.1图 14-1 表示的是有 3 个盘子的初始状态。

问题是，将盘子从第一根柱子移到第三根柱子上，并保持原来的次序。但必须遵守以下规则：

- 1) 一次移动一个盘子。每次只能移动最上面的盘子。
- 2) 盘子不能放在比自己小的盘子上面。
- 3) 在遵守前两条规则的同时，可以用第二根柱子暂存盘子。

解决方案是移动序列。例如，如果 3 个盘子在柱 1 上，则下述含 7 个移动步骤的序列将盘子移到柱 3 上，使用柱 2 作为临时柱：14.2



图 14-1 3 个盘子的汉诺塔的初始状态

将一个盘子从柱 1 移到柱 3

将一个盘子从柱 1 移到柱 2

将一个盘子从柱 3 移到柱 2

将一个盘子从柱 1 移到柱 3

将一个盘子从柱 2 移到柱 1

将一个盘子从柱 2 移到柱 3

将一个盘子从柱 1 移到柱 3

这些移动的过程如图 14-2 所示。

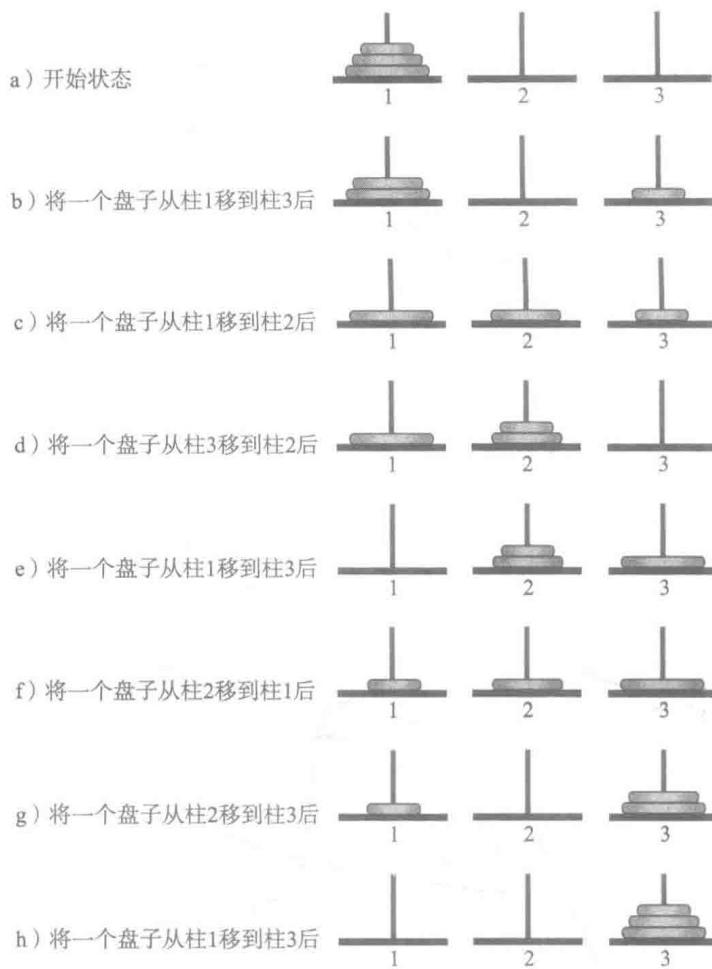


图 14-2 求解 3 个盘子的汉诺塔问题的移动序列



**学习问题 1** 我们通过反复试验，找到前面 3 个盘子的解决方案。使用同样的方法，找到求解 4 个盘子问题的移动序列。

对于 4 个盘子，问题的求解需要 15 步移动，通过反复试验来找答案有点困难。对于多于 4 个的盘子，找到求解方案要困难得多。我们需要的是一个能对任意多个盘子求解的算法。虽然通过反复试验求解很困难，但找到能给出答案的递归算法却相当简单。

### 旁白

在 19 世纪后期，伴随着一个传说出现了汉诺塔问题。据说，一群僧侣开始将 64 个盘子从一个塔上移动到另一个塔上。当他们完成时，世界将终结。当你读完本节，就会知道僧侣们——或是他们的后代——不可能完成。当他们这样做时，更有可能是盘子被磨穿，而不是世界将终结！

递归算法通过求解一个或多个更小的同类型问题来解决问题。这里，问题规模就是盘子的个数。假定第一个柱子上有 4 个盘子，如图 14-3a 所示，那么我要求你来求解这个问题。最终，你必须移动最下面的盘子，但必须先移动它上面的 3 个盘子。根据我们的规则要求朋友来移动那 3 个盘子——更小的问题，但让柱 2 作为目标柱。允许朋友将柱 3 作为备用柱。

图 14-3b 显示的是朋友工作的最终结果。

当你的朋友告诉你任务完成时，你将柱 1 上剩下的一个盘子移到柱 3 上。移动一个盘子是个简单的任务。你不需要帮忙——或递归——就能完成它。这个盘子是最大的盘子，所以它不能放在任何其他盘子的上面。所以在步移动之前柱 3 必须是空的。移动后，最大的盘子将是柱 3 上的第一个盘子。图 14-3c 表示的是你工作的结果。

现在要求一位朋友根据规则将柱 2 上的 3 个盘子移到柱 3 上。允许朋友将柱 1 作为备用柱。当你的朋友告诉你任务完成时，你可以告诉我，你的任务也完成了。图 14-3d 表示的是最终的结果。

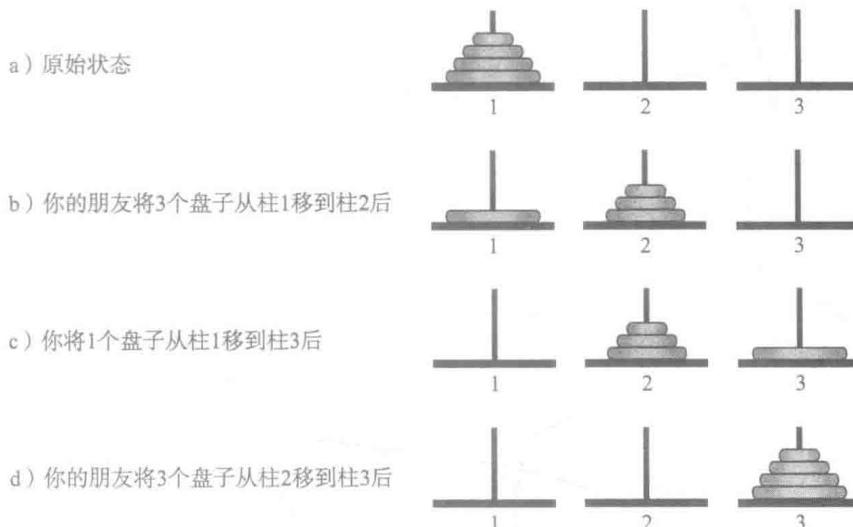


图 14-3 递归求解 4 个盘子中的更小问题

在写伪代码描述算法之前，必须定义基础情形。如果柱 1 上仅有一个盘子，则我们可以直接将它移到柱 3 上而不需要使用递归。将这种情形作为基础情形，算法如下所示。14.4

```
根据汉诺塔问题的规则，可以借助于tempPole，将numberOfDisks个盘子从startPole移到
endPole上的算法
if (numberOfDisks == 1)
 将盘子从startPole移到endPole
else
{
 将除最下面一个盘子之外的所有盘子从startPole移到tempPole
 将盘子从startPole移到endPole
 将所有盘子从tempPole移到endPole
}
```

此时，我们可以进一步将算法写为：

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
 将盘子从startPole移到endPole
else
{
 solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
 将盘子从startPole移到endPole
 solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

如果选择0个盘子代替1个盘子作为基础情形，则可以稍稍简化算法，如下所示。

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
// Version 2.
if (numberOfDisks > 0)
{
 solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
 将盘子从startPole移到endPole
 solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

虽然写起来更容易，但第二个版本的算法会执行更多次的递归调用。不过两个版本有相同的移动序列。



### 学习问题2 对于两个盘子，刚刚给出的算法的每个版本各需要多少次递归调用？

你对递归的了解能让你确信算法的两个版本都是正确的。递归能让我们解决一个看似困难的问题。但是这个算法有效率吗？如果我们使用迭代会不会更好些？

**14.5** 效率。让我们来看看算法的效率。当初始有 $n$ 个盘子时需要移动多少步？令 $m(n)$ 表示solveTowers求解 $n$ 个盘子时必需的移动步数。显然，

$$m(1) = 1$$

对于 $n > 1$ ，算法使用两次递归调用，每次调用解决 $n-1$ 个盘子的问题。每种情形中所需的移动步数是 $m(n-1)$ 。所以由算法有

$$m(n) = m(n-1) + 1 + m(n-1) = 2 \times m(n-1) + 1$$

从这个方程中，可以看到 $m(n) > 2 \times m(n-1)$ 。即求解有 $n$ 个盘子的问题比求解有 $n-1$ 个盘子的问题，需要多于两倍的移动步数。

看起来， $m(n)$ 与2的幂次有关。对于几个 $n$ 值，我们计算 $m(n)$ 的递推值：

$$m(1) = 1, m(2) = 3, m(3) = 7, m(4) = 15, m(5) = 31, m(6) = 63$$

似乎是

$$m(n) = 2^n - 1$$

我们可以使用数学归纳法证明这个猜想。如下。

**14.6** 归纳证明 $m(n) = 2^n - 1$ 。已知， $m(1) = 1$ 且 $2^1 - 1 = 1$ ，对 $n=1$ 猜想是正确的。现在假定，对于 $n = 1, 2, \dots, k$ ，它是正确的，考虑 $m(k+1)$ 。

$$\begin{aligned} m(k+1) &= 2 \times m(k) + 1 && (\text{使用递推关系}) \\ &= 2 \times (2^k - 1) + 1 && (\text{假定 } m(k) = 2^k - 1) \\ &= 2^{k+1} - 1 \end{aligned}$$

因为对于 $n=k+1$ ，猜想是正确的，所以对所有的 $n \geq 1$ ，它是正确的。

**14.7** 指数增长。求解汉诺塔问题所需的移动步数呈 $n$ 个盘子的指数增长。即 $m(n) = O(2^n)$ 。这个增长率令人害怕，你可以从下列 $2^n$ 的值了解到：

$$2^5 = 32$$

$$2^{10} = 1024$$

$$2^{20} = 1\,048\,576$$

$$2^{30} = 1\,073\,741\,824$$

$$2^{40} = 1\,099\,511\,627\,776$$

$$2^{50} = 1\,125\,899\,906\,842\,624$$

$$2^{60} = 1\,152\,921\,504\,606\,846\,976$$

还记得段 14.29 的结尾提到的僧侣吗？他们要移动  $2^{64}-1$  步。如果你想活着看到结果的话，显然这个指数阶算法仅能用于较小的  $n$  值。

在谴责并抛弃递归算法之前，必须明白你不能做得更好。不是你，也不是僧侣，不是任何人。接下来使用数学归纳法来说明这个结论。

证明汉诺塔问题的求解不能少于  $2^n-1$  步。已经看到，我们解决汉诺塔问题的算法需要  $m(n) = 2^n-1$  步。因为知道至少存在一个算法——我们已经找到了——那就一定存在一个最快的算法。令  $M(n)$  表示这个优化算法移动  $n$  个盘子时需要的步数。我们来说明对于  $n \geq 1$ ，有  $M(n)=m(n)$ 。14.8

当问题仅有一个盘子时，我们的算法需要移动一步就可解决。我们不能做得更好，所以有  $M(1)=m(1)=1$ 。如果假定  $M(n-1) = m(n-1)$ ，考虑  $n$  个盘子。回看图 14-3b，你能看到在我们的算法中有一个时刻，最大的盘子留在一根柱子上，而其余的  $n-1$  个盘子在另一根柱子上。这个状态对优化算法也必须为真，因为没有其他办法来移动最大的盘子。所以优化算法必须在  $M(n-1) = m(n-1)$  步内将这  $n-1$  个盘子从柱 1 移动到柱 2。

移动了这个最大的盘子后（图 14-3c），优化算法又花另外的  $M(n-1) = m(n-1)$  步将  $n-1$  个盘子从柱 2 移到柱 3。优化算法总共进行了  $2 \times M(n-1)+1$  步。所以

$$M(n) \geq 2 \times M(n-1) + 1$$

现在使用假设条件  $M(n-1) = m(n-1)$ ，然后再使用段 14.5 中所给的  $m(n)$  的递推关系，得到

$$M(n) \geq 2 \times m(n-1) + 1 = m(n)$$

我们刚证明了  $M(n) \geq m(n)$ 。但因为优化算法不会比我们的算法移动更多的步数，表达式  $M(n) > m(n)$  不能成立。所以对于所有的  $n \geq 1$ ，必须有  $M(n)=m(n)$ 。

**迭代算法。**找到求解汉诺塔问题的迭代算法，不像找递归算法这样简单。我们知道，任何迭代算法需要的移动步数至少与递归算法一样多。迭代算法将节省记录递归调用的开销——空间上的和时间上的，但它不是真的比 `solveTowers` 更高效。下节将讨论使用迭代和递归求解汉诺塔问题的算法，本章结尾的项目 2 将讨论使用完全迭代算法求解问题。14.9

让我们来替换段 14.4 中所给的 `solveTowers` 算法的第二个版本中的尾递归。14.10

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks > 0)
{
 solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
 将盘子从startPole移到endPole
 solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

算法含有两个递归调用。第二个是尾递归，因为它是算法的最后一个动作。所以我们可以试着将第二个递归调用替换为相应的赋值语句，并使用循环来重复方法的逻辑，包括第一个递归调用，如下所示。

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
while (numberOfDisks > 0)
{
 solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
 将盘子从startPole移到endPole
```

```
 numberOfDisks = numberOfDisks - 1
 startPole = tempPole
 tempPole = startPole
 endPole = endPole
}
```

但是这个算法并不完全正确，显然，将 `endPole` 赋给自己是多余的。将 `tempPole` 赋给 `startPole`，然后将 `startPole` 赋给 `tempPole` 又破坏了 `startPole` 的值，而 `tempPole` 值并没有改变。我们需要做的是交换 `tempPole` 和 `startPole` 的值。来看看到底发生了什么。

真正移动盘子的唯一指令是“将盘子从 `startPole` 移到 `endPole`”。这条指令移动尚不在 `endPole` 上的最大的盘子。要移动的这个盘子在柱子的最下面，所以它上面的所有盘子都必须先移走。这些盘子由第一个递归调用来移动。如果我们想省掉第二个递归调用，那么在重复第一个递归调用之前，我们要做什么呢？必须确保尚未移到 `endPole` 上的盘子在 `startPole` 上。而那些盘子在 `tempPole` 上，这是第一个递归调用的结果。所以我们必须交换 `tempPole` 和 `startPole` 的内容。

做了这些修改后得到下面改后的算法：

```

Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
while (numberOfDisks > 0)
{
 solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
 将盘子从startPole移到endPole
 numberOfDisks--
 交换tempPole和startPole的内容
}

```

这个修改后的算法与众不同，它的循环中含有一个递归调用。当 `numberOfDisks` 为 0 时，发生这个递归调用的基础情形。即使方法中不含有一个 `if` 语句，它也检测基础情形，终止递归调用。

简单问题的低劣求解方案

如你所见，递归方案是非常有用的，但有些递归方案的效率低下，你应该避免使用。我们现在要讨论的问题是简单的，常出现在数学计算中，且有递归求解方法，很自然地你可能会冒险去用。不要！



示例：Fibonacci 数。早在 13 世纪，数学家 Leonardo Fibonacci(列奥纳多·斐波那契)提出一个模拟一对兔子后代数量的整数序列。后来命名为 Fibonacci 数列 (*Fibonacci sequence*)，这些数出人意料地用在许多应用中。

$$F_0 = 1$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{当 } n \geq 2 \text{ 时}$$

你能明白下列递归算法为什么是生成这个数列的一个诱人地方法。

*Algorithm* Fibonacci(n)  
if (n <= 1)

```

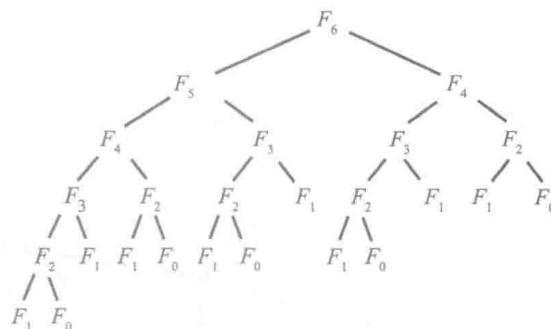
 return 1
else
 return Fibonacci(n - 1) + Fibonacci(n - 2)

```

这个算法进行两次递归调用。事实上这本身并不困难。之前你见过完美的算法——第9章段9.18中的displayArray和段14.4中的solveTowers——都进行了多次递归调用。这里的麻烦是，重复进行了相同的递归调用。调用 `Fibonacci(n)` 时将调用 `Fibonacci(n-1)`，然后调用 `Fibonacci(n-2)`。但对 `Fibonacci(n-1)` 的调用又必须计算 `Fibonacci(n-2)`，所以对同一个 `Fibonacci` 数计算了两次。

事情变得更糟。调用 `Fibonacci(n-1)` 时也会调用 `Fibonacci(n-3)`。前面两次调用 `Fibonacci(n-2)` 时每次都会调用 `Fibonacci(n-3)`，所以 `Fibonacci(n-3)` 被计算了3次。图14-4a说明了  $F_6$  与前面 `Fibonacci` 数的相依关系，也表明一个具体的数被 `Fibonacci` 方法重复计算的次数。相反，图14-4b表示迭代计算  $F_6$  的过程，前面的每项只计算一次。很显然递归方案的效率低下。下一段将说明它的效率是多么低。

a) 递归地  
 $F_2$  被计算5次  
 $F_3$  被计算3次  
 $F_4$  被计算2次  
 $F_5$  被计算1次  
 $F_6$  被计算1次



b) 迭代地  
 $F_0 = 1$   
 $F_1 = 1$   
 $F_2 = F_1 + F_0 = 2$   
 $F_3 = F_2 + F_1 = 3$   
 $F_4 = F_3 + F_2 = 5$   
 $F_5 = F_4 + F_3 = 8$   
 $F_6 = F_5 + F_4 = 13$

图14-4 `Fibonacci` 数  $F_6$  的计算

算法 `Fibonacci` 的时间效率。可以使用段9.22到段9.27给出的递推关系式研究 `Fibonacci` 算法的效率。首先，注意到  $F_n$  需要一次加法操作再加上  $F_{n-1}$  和  $F_{n-2}$  的需求。所以，如果  $t(n)$  表示计算  $F_n$  的算法的时间需求，则有

$$\begin{aligned} t(n) &= 1 + t(n-1) + t(n-2) \quad \text{对于 } n \geq 2 \\ t(1) &= 1 \\ t(0) &= 1 \end{aligned}$$

这个递推关系很像是 `Fibonacci` 数自己。对于  $t(n)$  与 `Fibonacci` 数相关这件事，你不应该感到意外。实际上，如果看图14-4a，并统计 `Fibonacci` 数从  $F_2$  到  $F_6$  出现的次数，就会得到 `Fibonacci` 数列。

为找到  $t(n)$  与  $F_n$  之间的关系，我们使用几个  $n$  值来展开  $t(n)$ :

$$\begin{aligned} t(2) &= 1 + t(1) + t(0) = 1 + F_1 + F_0 = 1 + F_2 > F_2 \\ t(3) &= 1 + t(2) + t(1) > 1 + F_2 + F_1 = 1 + F_3 > F_3 \\ t(4) &= 1 + t(3) + t(2) > 1 + F_3 + F_2 = 1 + F_4 > F_4 \end{aligned}$$

14.12

14.13

我们猜测对于  $n \geq 2$ , 有  $t(n) > F_n$ 。注意到  $t(0) = 1 = F_0$  且  $t(1) = 1 = F_1$ 。这些不满足我们猜测的严格不等式。

现在证明我们的猜测确实是正确的。(第一次阅读时你可以跳过证明部分。)

**14.14 使用归纳法证明**, 对于  $n \geq 2$ , 有  $t(n) > F_n$ 。因为  $t(n)$  的递推关系含有两个递归项, 所以我们需要两个基础情形。在前一段中, 已经知道  $t(2) > F_2$  且  $t(3) > F_3$ 。现在, 如果对于  $n = 2, 3, \dots, k$ , 有  $t(n) > F_n$ , 则我们只需表明  $t(k+1) > F_{k+1}$  即可。可以像下面这样做。

$$t(k+1) = 1 + t(k) + t(k-1) > 1 + F_k + F_{k-1} = 1 + F_{k+1} > F_{k+1}$$

故有结论, 对于  $n \geq 2$ , 有  $t(n) > F_n$ 。

因为知道对于  $n \geq 2$ , 有  $t(n) > F_n$ , 所以可以说  $t(n)$  是  $\Omega(F_n)$  的。回忆第 4 章, 大  $\Omega$  符号意味着,  $t(n)$  至少是与 Fibonacci 数  $F_n$  同样大。事实上我们可以直接计算  $F_n$ , 而不需要使用段 14.11 给出的递推关系。可以证明

$$F_n = (a^n - b^n) / \sqrt{5}$$

其中,  $a = (1 + \sqrt{5}) / 2$ , 且  $b = (1 - \sqrt{5}) / 2$ 。因为  $|1 - \sqrt{5}| < 2$ , 有  $|b| < 1$  且  $|b^n| < 1$ 。因此有

$$F_n > (a^n - 1) / \sqrt{5}$$

所以,  $F_n = \Omega(a^n)$ , 因为已经知道  $t(n) = \Omega(F_n)$ , 所以有  $t(n) = \Omega(a^n)$ 。通过计算前面的表达式可知,  $a$  的值约等于 1.6。我们得到结论,  $t(n)$  以  $n$  的指数增长。即递归计算  $F_n$  所需的时间按  $n$  的指数增长。

**14.15** 本节的开头, 我们观察到每个 Fibonacci 数都是数列中前两个 Fibonacci 数的和。这个观察能让我们找到  $O(n)$  的迭代方案。(见本章最后的练习 1。) 虽然递归方案清晰简单, 使得它成为诱人的选择, 但使用时效率太低。



**程序设计技巧:** 不要使用在递归调用中重复解决同一问题的递归方案。



**学习问题 3** 如果用递归方式计算 Fibonacci 数  $F_6$ , 那么需要执行多少次递归调用?  
需要执行多少次加法?

**学习问题 4** 如果用迭代方式计算 Fibonacci 数  $F_6$ , 那么需要执行多少次加法?

## 语言和语法

**14.16** 你知道像英语这样的人类语言及像 Java 这样的编程语言。语言 (language) 只是一组遵守某些规则的字符序列。这些规则组成语言的语法 (grammar)。通常, 计算机科学家使用一些特殊的符号来写语法规则:

- $x|y$  表示  $x$  或  $y$ 。
- $x \cdot y$  或更简单的  $xy$  表示  $x$  后面是  $y$ 。
- $<s>$  表示  $s$  的任何实例, 其中  $s$  必须是语法中定义的一个符号。

## Java 标识符语言

**14.17** Java 中的标识符是从字母、数字和美元符号中选出的一个字符序列。标识符不能是数字开头, 也不能含有空格。我们可以将所有可能的 Java 标识符组成一个语言, 可以写下面的

语法来定义这个语言：

```
<标识符>=<字母>|<标识符><字母>|<标识符><数字>
<字母>=a|b|...|z|A|B|...|Z|$
<数字>=0|1|...|9
```

这个定义的含义如下，其中美元符号看作一个字母：

“一个标识符可以是一个字母、一个标识符后跟一个字母，或是一个标识符后跟一个数字。”

注意，标识符又出现在它自己的定义中。这个语法是递归的，与许多语法一样。递归语法常能让你写一个简单的递归算法，检测给定的字符串是否属于语言。这样的一个算法称为语言的识别算法（recognition algorithm）。

我们通过检查前面的语法，来写 Java 标识符语言的识别算法。如果字符串  $s$  满足下面的两个条件之一，则  $s$  是一个 Java 标识符：

- $s$  的长度是 1，且  $s$  是一个字母。（这个陈述像是基础情形。）
- $s$  的长度大于 1，其最后面的字符是字母或数字，且  $s$  去掉最后面的字符后仍是一个标识符。

从这个逻辑出发，可以为递归方法写下面的伪代码，测试一个字符串是否是一个合法的 Java 标识符，即在 Java 标识符语言中：

```
// Returns true if s is a valid Java identifier; otherwise returns false.
isIdentifier(s: string): boolean
{
 if (s 的长度是 1) // Base case
 {
 if (s 是一个字母)
 return true
 else
 return false
 }
 else if (s 的最后面的字符是字母或数字)
 return isIdentifier(s 去掉最后面的字符)
 else
 return false
}
```



**学习问题 5** 对下列每个字符串跟踪伪代码 `isIdentifier`。

- AB5
- A?B

**学习问题 6** 回文是一个从左向右读与从右向左读都一样的字符串。例如 `level`、`noon` 和 `racecar` 都是回文，但 `racecars` 不是。写一个单字回文语言的语法。

**学习问题 7** 写回文语言识别算法的伪代码。

## 前缀表达式语言

回忆第 5 章对代数表达式的讨论。像  $a + b * c$  这样的普通表达式称为中缀表达式，其中二元运算符放在其两个操作数之间。为避免歧义，中缀表达式依赖于优先级规则、结合律和括号。例如，中缀表达式  $a + b * c$  中  $+$  的第二个操作数是  $(b * c)$ ，且  $*$  的第一个操作数是  $b$ ，而不是  $(a + b)$ 。 $*$  比  $+$  的优先级更高的规则，消除了二义性。如果你想要另一种解释，则必

须使用括号：

$$(a + b) * c$$

即使有优先级规则，像

$$a / b * c$$

这样的表达式的含义也是不明确的。一般地，/ 和 \* 有相等的优先级，所以你可以将表达式解释为

$$(a / b) * c$$

或解释为

$$a / (b * c)$$

常见的做法是从左至右结合，所以得出第一种解释。

第5章定义了前缀表达式和后缀表达式，即使它们从不使用优先级规则、结合律和括号，其含义也没有歧义。在前缀表达式中，每个二元运算符放在它的操作数之前。例如， $+ a b$  是前缀表达式，它等价于中缀表达式  $a + b$ ，而中缀表达式  $(a + b) * c$  的前缀形式是  $* + a b c$ 。

14.20

手工将全括号的中缀表达式转换为前缀形式。如果中缀表达式是全括号的，则你可以执行一个简单的手工计算将它转换为前缀形式或后缀形式。因为每个运算符对应于一对括号，所以你只需将运算符移到其相应的开括号“(”标记的位置。这个位置在运算符的操作数之前。然后删除所有的括号。

例如，考虑下面的全括号中缀表达式

$$((a + b) * c)$$

要将这个表达式转为前缀形式，首先应将每个运算符移到其相应的开括号标记的位置：

$$\begin{array}{c} ( (a b) c ) \\ \downarrow \quad \downarrow \\ * \quad + \end{array}$$

接下来，删除括号得到所要的前缀表达式：

$$* + a b c$$

当中缀表达式不是全括号形式时，转为前缀形式或是后缀形式更复杂一些。回忆第5章使用栈将中缀表达式转为后缀形式的过程。



**学习问题 8** 写出代表下列中缀表达式的前缀表达式：

$$(a / b) * c - (d + e) * f$$

**学习问题 9** 写出代表下列前缀表达式的中缀表达式：

$$- a / b + c * d e f$$

**学习问题 10** 下列字符串是前缀表达式吗？为什么？

$$+ - / a b c * + d e f * g h$$

**学习问题 11** 要将全括号中缀表达式转为后缀形式，应该将每个运算符移到其相应的闭括号“)”标记的位置。这个位置是运算符的操作数之后。然后删除所有的括号。将全括号中缀表达式  $((a + b) * c)$  转为后缀形式。

14.21

因为前缀和后缀表达式不使用优先级规则、结合律规则和括号，所以它们的语法更简单。定义所有前缀表达式语言的语法是：

< 前缀表达式 > = < 操作数 > | < 运算符 > < 前缀表达式 > < 前缀表达式 >

< 操作数 > = a | b | ... | z

< 运算符 > = + | - | \* | /

注意，这个语法是递归的。

由这个语法，可以推断字符串  $s$  是一个前缀表达式，当且仅当

- $s$  的长度是 1，且它是一个小写字母

或者

- $s$  的长度大于 1，且它的形式是 < 运算符 > < 前缀表达式 > < 前缀表达式 >

可以从这些陈述写一个递归识别算法。

前段中第一个着重号描述的情形很容易检查。但测试第二个着重号中的情形有点微妙。14.22  
如何能测试两个连续的前缀表达式？若要查看是否有两个连续的前缀表达式，必须先识别出第一个前缀表达式。如果在这个前缀表达式的后面添加任一个非空格字符，则得到的不再是前缀表达式。（本章结尾处的练习 10 要求你证明这个陈述。）所以，如果第一个前缀表达式的结束位置是  $p$ ，则应该从位置  $p+1$  处开始查看第二个前缀表达式。如果找到了第二个表达式，则应该到达字符串的结尾处。

 注：如果  $P$  是前缀表达式而  $S$  是任意的非空格字符的非空字符串，则  $PS$  不会是前缀表达式。

示例。使用前面的思想来说明  $+*ab - cd$  是一个前缀表达式。因为字符串以 + 开头，所以需要说明它有形式  $+E_1E_2$ ，其中  $E_1$  和  $E_2$  是前缀表达式。即14.23

$$+*ab - cd = +E_1E_2$$

现在  $E_1$  以运算符 \* 开头，所以  $E_1$  应该是一个前缀表达式，它的形式必须是

$$E_1 = * E_3 E_4, \text{ 其中}$$

$$E_3 = a$$

$$E_4 = b$$

因为  $E_3$  和  $E_4$  是前缀表达式，所以  $E_1$  是前缀表达式。类似地，可以写出

$$E_2 = - E_5 E_6, \text{ 其中}$$

$$E_5 = c$$

$$E_6 = d$$

可以看出  $E_2$  是前缀表达式。

可以写一个方法来测试一个字符串是否是前缀表达式，先构造一个递归的值方法 `endPre(str, first)`，找到字符串  $str$  中从位置  $first$  开始的第一个前缀表达式的结尾。如果成功，则方法返回前缀表达式结尾的下标。如果没有这样的前缀表达式存在，则 `endPre` 返回 -1。方法的伪代码如下：14.24

```
// Returns either the index of the last character in the prefix expression that
// begins at index first of str or -1, if no such prefix expression exists.
// Precondition: The string str contains no blank characters.
endPre(str: string, first: integer): integer
{
 last = str.length() - 1
 if (first < 0 或 first > last)
 return -1
}
```

```

ch = str 中位于 first 位置的字符
if (ch 是一个操作数)
 return first // Index of last character in simple prefix expression
else if (ch 是一个运算符)
{
 // Find the end of the first prefix expression
 end1 = endPre(str, first + 1)

 // If the end of the first prefix expression was found, find the end of the
 // second prefix expression
 if (end1 > -1)
 return endPre(str, end1 + 1)
 else
 return -1
}
else
 return -1
}

```

**学习问题 12 跟踪 endPre("/\*ab-cd", 0) 的执行。**



14.25

现在可以将 endPre 用在前缀表达式的识别算法中了，如下所示。

```

// Returns true if a string is a prefix expression; otherwise returns false.
// Precondition: The string str contains no blank characters.
isPrefix(str: string): boolean
{
 strLength = str 的长度
 end1 = endPre(str, 0)
 return (end1 >= 0) 且 (end1 == strLength - 1)
}

```

## 前缀表达式的计算

14.26

给定一个前缀表达式，如何对它求值？因为前缀表达式中的每个运算符都位于其两个操作数的前面，所以可以从运算符开始在表达式中向前查看。但是，这样的操作数本身可能又是前缀表达式，我们必须要先对它求值。这些前缀表达式是原表达式中的子表达式，所以必须是“更小的”。这个问题的递归求解方案似乎是自然的。

下列伪代码方法对前缀表达式求值，所有单字符操作数都给了数值。这个算法比第 5 章给出的对中缀表达式求值的算法更简单。

```

// Returns the value of a given prefix expression.
// Precondition: str is a string containing a valid prefix expression with no
// blanks.
evaluatePrefix(str: string): real
{
 strLength = str 的长度
 if (strLength == 1)
 return str 中单字符操作数的值 // Base case
 else
 {
 op = str.charAt(0) // str begins with an operator

 // Find the end of the first prefix expression, that is, the first operand
 endFirst = endPre(str, 1)

 // Recursively evaluate this first prefix expression
 operand1 = evaluatePrefix(str[1..endFirst])
 }
}

```

```

// Find the end of the second prefix expression, that is, the second
// operand
endSecond = strLength - endFirst + 1

operand2 = evaluatePrefix(str[endFirst + 1..endSecond])

// Evaluate the prefix expression
return operand1 op operand2
}
}

```

## 间接递归

有些递归算法不直接进行递归调用。例如，可能有下列一连串的事件：方法 A 调用方法 B，方法 B 调用方法 C，而方法 C 又调用方法 A。这样的递归——称为间接递归（indirect recursion）——更难理解并跟踪，但在某些应用中自然存在。14.27

例如，下列规则描述了稍微受限的合法的中缀形式代数表达式。

- 一个代数表达式或者是一个项，或者是由运算符 + 或 – 隔开的两个项。
- 一个项或者是一个因子，或者是由运算符 \* 或 / 隔开的两个因子。
- 一个因子或者是一个变量，或者是一个包含在圆括号内的代数表达式。
- 一个变量是一个单字符。

假定方法 `isExpression`、`isTerm`、`isFactor` 和 `isVariable` 分别检测一个字符串是否是表达式、项、因子及变量。方法 `isExpression` 调用 `isTerm`，后者又调用 `isFactor`，而它又调用 `isVariable` 和 `isExpression`。图 14-5 说明了这些调用。

间接递归的一个特殊情况，即方法 A 调用方法 B，而方法 B 调用方法 A，称为相互递归（mutual recursion）。本章最后的项目 8 描述了相互递归的一个示例。

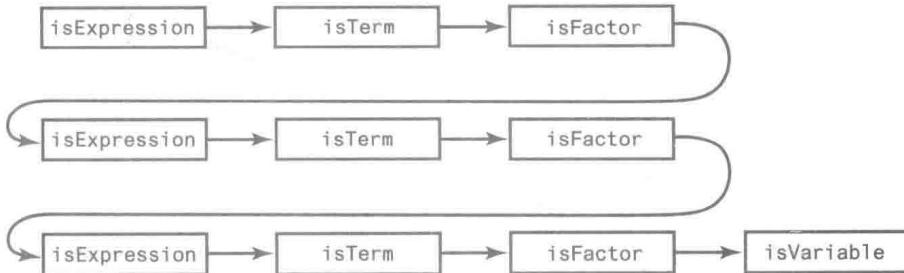


图 14-5 间接递归示例



**学习问题 13** 写出由段 14.27 给出的 4 个规则所定义的表达式语言的语法。

**学习问题 14** 使用回答学习问题 13 时所写的语法，说明字符串  $a + b * c$  属于这个中缀表达式语言，而  $a / b * c$  不属于这个语言。

**学习问题 15** 如何修改字符串  $a / b * c$ ，让它属于段 14.27 给出的 4 条规则所定义的语言？

**学习问题 16** 段 14.27 给出的 4 条规则以何种方式限制了中缀表达式？

## 回溯

回溯是另一个问题求解策略，而且常常与递归同时使用。回溯（backtrack）意味着回退。14.28

自己的脚步。当你必须在解决问题的几种潜在方案中进行选择时，尝试一个并看看你是否能成功。如果成功了，则任务完成。但如果你的选择不成功，则退回到更早的状态并尝试另一种可能的方案。注意，你仅回退有必要回退的步骤。即回退到最近的提供另一个选择的点。如果那个点不能提供其他的选择了，则必须回退到更远的下一个最早的决策点。



**注：**强力破解技术 (brute-force technique) 检查一个解决方案中所有可能的选择。例如，考虑如第11章段11.13中给出的数组中的顺序查找。为确信给定的项不出现在指定数组中，顺序查找必须检查数组中的每个项。这个过程是穷举查找 (exhaustive search) 的一个示例，并且用到了强力破解。虽然回溯有一些强力破解，但它系统地消除了必须测试的许多选择。

## 穿越迷宫

14.29

维多利亚时代英国园林和现代玉米田中都已经建造了迷宫。一个典型的迷宫，如图14-6a所示，有一条或多条从入口通向出口的路径。但从入口开始的其他路径通向死胡同而不是通向出口。你能找到穿越这个迷宫的通路吗？

图14-6b展示了迷宫上的栅格。栅格中的每个小方格表示你能走的一步。当你穿越迷宫时，应该标记出你的轨迹。在迷宫的任一点上，检查你是否到达出口。如果到达了，则已经完成任务。否则走到相邻的未访问过的方格中，并继续从那里搜索，以尝试找到通向出口的路径。

例如，假定迷宫中你当前位置正南方的方格是空的，并且之前没有被访问过。你可以向南走一步，并尝试找到通向出口的路径。如果路径存在，则将位置标记为位于路径中。否则，找不到路径，则将位置标记为已访问并回溯。换句话说，你向北走一步，并继续查找，比如向东走一步。

下面的伪代码描述了查找成功路径的算法，从指定的方格开始：

```

searchFrom(currentSquare) : boolean
{
 success = false
 if (currentSquare在迷宫内、是空的且未被访问)
 {
 标记currentSquare为已访问
 if (currentSquare是出口)
 success = true
 else
 {
 success = searchFrom (currentSquare下面的方格) // Try south
 if (!success)
 success = searchFrom (currentSquare东面的方格) // Backtrack
 and try east
 if (!success)
 success = searchFrom (currentSquare北面的方格) // Backtrack
 and try north
 if (!success)
 success = searchFrom (currentSquare西面的方格) // Backtrack
 and try west
 }
 }
}

```

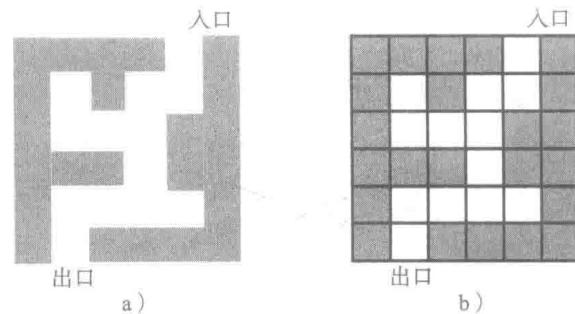


图14-6 有一个入口和一个出口的二维迷宫

```

 }
 if (success)
 将 currentSquare 标记为到出口的路径的一部分
 }
 return success
}

```



### 设计决策：应该使用回溯吗？

使用回溯的方案像是“强力破解”方法，其中每种可能的情形都要尝试，看看它是不是解决方案。这样的算法在时间上可能极其低效。但是，你或许没有任何其他的方案。如果回溯的性能是好的——比如当要测试的可能情形比较少——回溯算法就是可以接受的。

你或许对一个特定问题的任何解决方法都感到满意，但可能想找到最优解。当然，“最优”的含义是有情境的。例如对于迷宫问题，最优路径可能是最短路径。在前面的算法 searchFrom 中，我们可以统计方案中的步数。还必须确保已经检查了所有可能的路径，测量了每条路径的长度，并跟踪迄今为止遇到的最短路径。以这种方式就可以忽略掉部分路径，只要它超出当前最小长度。

## $n$ 皇后问题

$n$  皇后问题使用一个棋盘，但不需要知道游戏的规则。只需要知道一个棋子，皇后。皇后是最强的棋子，因为它可以在水平方向、垂直方向或对角线上任意地走。本问题要求你将  $n$  个皇后放到  $n \times n$  的棋盘中，让它们都不受攻击。所以，每个皇后在其所在的行、列和对角线上都是唯一的。例如，图 14-7 展示的是 4 皇后问题的一个解答。这些问题的解答不一定是唯一的。

### 旁白

$n$  皇后问题是 1869 年提出的，研究不同  $n$  的求解方案。但是，在 1848 年之前，8 皇后问题就提出来了，因为一个典型的棋盘有 8 行和 8 列。1850 年出现了第一个解决方案，不久之后所有 92 种解决方案均已公布。对于 19 世纪的数学家来说，找到这些方案是一项艰巨的任务，因为将 8 皇后放在棋盘上可以有 40 亿种方法。但通过排除一些排列方式，即一行或一列中有多于一个皇后，数学家可以集中研究  $8!$  或 40320 种可能的对角线攻击情况。

虽然使用强力破解找到 4 皇后问题的一个解答并不困难，但是可以形式化一个可解答更大  $n$  值的方案。假定我们在  $4 \times 4$  棋盘的每列都放一个皇后。从第 1 列的第一个方格开始。图 14-8a 展示了这个皇后，并用箭头和阴影表示了它的攻击范围。在后面的各列中，不能再把另一个皇后放到阴影方格中。

当考虑第 2 列时，因为第 1 行含有一个皇后所以排除第一个方格，因为对角线攻击所以排除第 2 个方格，最后将 1 个皇后放到第 2 列的第 3 个方格内，如图 14-8b 所示。

当继续以这种方式放置皇后时，注意到，第 3 列的每个方格都在已有的两个皇后的攻击

14.30

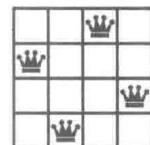


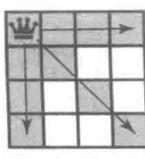
图 14-7 4 皇后问题的解答

14.31

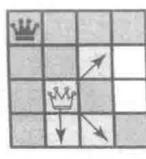
14.32

范围内。所以，因为我们不能在第3列放置皇后，故必须回溯——即后退——到第2列，将这列的皇后移到本列下一个可能的方格内。这个方格在最后一行，如图14-8c所示。当再次考虑第3列时，将皇后放在本列唯一可用的方格内，即第2行的方格，如图14-8d所示。

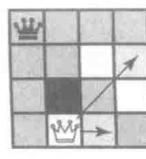
放置的3个皇后可以攻击第4列的所有方格，所以不能在第4列放置皇后。而是必须回溯到第3列，并试着移动这列的皇后。这是不可能的，从图14-8e中可以明白这一点，所以回溯到第2列。但是不能在第2列移动皇后（图14-8f），所以回退到第1列。如图14-8g所示，可以将第1列的皇后移到第2行。然后再次考虑第2列，将皇后放到第4行（图14-8h）。继续这个过程，在第3列和第4列都找到一个安全的方格，将皇后放置到位（如图14-8i和图14-8j所示）。



a) 第1列的  
第1个皇后



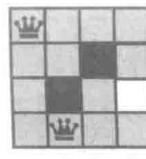
b) 第2列的第2个皇后。  
第3列的所有位置都在  
攻击范围内。



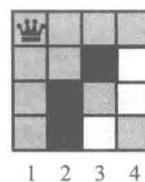
c) 回溯到第2列，  
尝试在另一个方  
格内放置皇后。



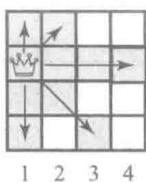
d) 第3列的第3个皇后。  
第4列的所有位置都在  
攻击范围内。



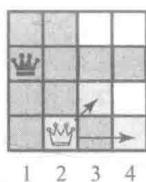
e) 回溯到第3列。  
皇后没办法移动。



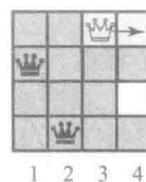
f) 回溯到第2列。  
皇后没办法移动。



g) 回溯到第1列，  
尝试在另一个方  
格内放置皇后。



h) 第2列的第2个皇后。 i) 第3列的第3个皇后。 j) 第4列的第4个皇后。  
这是答案！



■ = 可被现有皇后攻击的位置    □ = 可被新放置的皇后攻击的位置    ■ = 回溯过程中被拒绝的位置

图14-8 每列每次放置一个皇后求解4皇后问题



**学习问题 17** 手工找出4皇后问题的所有答案。

**学习问题 18** 手工找出5皇后问题的一个答案。

#### 14.33

在刚刚描述的处理中如何使用递归？我们的问题从n列开始。在成功将一个皇后放置在一列中后，考虑下一列。即我们解决有更少列的同一问题，这就是递归步骤。所以，从n列开始，每个递归步骤中考虑列数减1的更小的问题，当考虑没有列的问题时到达基础情形。

这个方案似乎满足递归求解的标准。但是，我们不知道是不是能在当前列成功地放置一个皇后。如果可以，则递归地考虑下一列。如果不能在当前列放置皇后，则必须回溯，正如已经描述过的。

最终，如果能在最后一列正确地放置一个皇后，将得到一个答案。但如果已经回溯到第1列且没有其他的方格可尝试时，则不存在答案。出现的这些情况中的每一种都是结束递归的基本情形。

下列伪代码描述了从给定列开始，在连续的各列中每列放置一个皇后的一个算法。

```

// Places one queen in each of consecutive columns beginning with a given one.
// Precondition: Previous columns contain queens that cannot attack one another.
placeQueens(column: integer): boolean
{
 if (column > number of columns)
 问题解决
 else
 {
 while (在给定的列中还存在未考虑的行且问题尚未解决)
 {
 在给定列找到不受之前各列的皇后攻击的下一行

 if (存在这样的一行)
 {
 在本行本列放置一个皇后

 // 尝试在下一列放置皇后
 if (!placeQueens(column + 1))
 {
 // 不可能在下一列放置皇后
 将刚刚放到棋盘上的皇后移动到本列的下一行
 }
 else
 {
 // 问题解决
 return true
 }
 }
 // 无解
 删除刚刚放置的皇后
 return false
 }
 }
}

```

使用棋盘左上角的新皇后调用 placeQueens，开始求解：

```

solveQueens()
{
 placeQueens(firstColumn)
}

```

solveQueens 执行完毕，如果找到答案则可以显示棋盘。

## 本章小结

- $n$  个盘子的汉诺塔问题的求解至少需要  $2^n - 1$  次移动。这个问题的递归方案清晰，而且尽可能高效。但作为一个  $O(2^n)$  的算法，它只适用于很小的  $n$  值。
- Fibonacci 数列中头两个数之后的每个数，都是其前两个值之和。递归计算 Fibonacci 数的效率极其低下，因为所需的之前的每个数都要计算很多次。
- 语言是字符串的集合。这些字符串符合一定的规则，即语言的语法。根据这个语法，可以创建识别算法，判定所给字符串是否在语言中。语法和识别算法常常是递归的，所以能简明地描述大量的语言。
- 中缀代数表达式比前缀形式或后缀形式都更常见且更易用。但是中缀表达式需要括号和优先级规则及结合律来消除歧义。结果，中缀表达式语言有一个复杂的语法。另一方面，前缀和后缀表达式不需要使用括号或优先级规则及结合律。它们的语法比中缀表达式的语法简单很多。

- 当一个方法调用一个方法，后者又调用一个方法，……，直到又调用第一个方法时，产生间接递归。
- 回溯是一种采用递归及猜测序列的解决策略，最终导向一个解决方案。如果某个猜测导向死胡同，则按逆序退回步骤，替换猜测，并尝试再次去求解。

## 练习

- 段 14.11 介绍了 Fibonacci 数列。递归计算这个数列是低效的，要花太多的时间。写两个方法使用迭代代替递归来计算 Fibonacci 数列中的第  $n$  项。一个方法将使用数组保存 Fibonacci 数。另一个方法将使用 3 个变量保存数列中的当前值和它前面的两个值。  
每个迭代方法的大  $O$  是多少？将这些结果与递归算法的性能进行比较。
- 对于 3 个盘子，段 14.4 中给出的两个 `solveTowers` 算法各有多少次递归调用？
- 考虑由下列语法定义的语言：
 

```
<字>=<破折号>|<点>|<字>|<字>|<破折号>
<点>=·
<破折号>=-
```

  - 写出本语言中的所有 3 字符串。
  - 串 ----- 在这个语言中吗？解释之。
  - 写一个本语言中破折号比点多的 7 字符串。展示你是如何知道你的答案是正确的。
  - 写递归的识别算法 `isIn(str)` 的伪代码，如果字符串 `str` 在本语言中则返回真，否则返回假。
- 考虑由下列语法定义的语言：
 

```
<字>=$|a|<字>a|b|<字>b|···|z|<字>z
```

  - 写出本语言中的所有 3 字符串。
  - 空字符串在本语言中吗？
  - `abc$cba` 在本语言中吗？
  - `cba$abc` 在本语言中吗？
  - `abc$abc` 在本语言中吗？
- 写后缀表达式语言的一个语法。
- `ab/c*c*efg*h/+d--` 是后缀表达式吗？根据你在练习 5 中给出的语法进行解释。
- 写全括号中缀表达式语言的一个语法。
- 考虑下列字符串语言：字母 A、字母 B、字母 C 后跟语言中的一个字符串、字母 D 后跟语言中的一个字符串。例如，A、B、CA、CB、CCA、CCB、DA、DB、DCA 和 DCB 都在语言中。
  - 写这个语言的一个语法。
  - `CAB` 在本语言中吗？解释之。
- 跟踪 `endPre("*/+abcd", 0)` 的执行。
- 证明下列单字母操作数：如果  $E$  是前缀表达式而  $Y$  是非空格字符组成的非空字符串，则  $E Y$  不是合法的前缀表达式。（提示：对  $E$  的长度进行归纳证明。）
- 使用伪代码，描述段 14.27 描述的 `isExpression`、`isTerm`、`isFactor` 和 `isVariable` 各方法的逻辑。
- 使用段 14.29 给出的算法 `searchFrom`，找到并画出图 14-9 所示的迷宫中从入口到出口的一条路径。
- 找出 6 皇后问题的所有解。

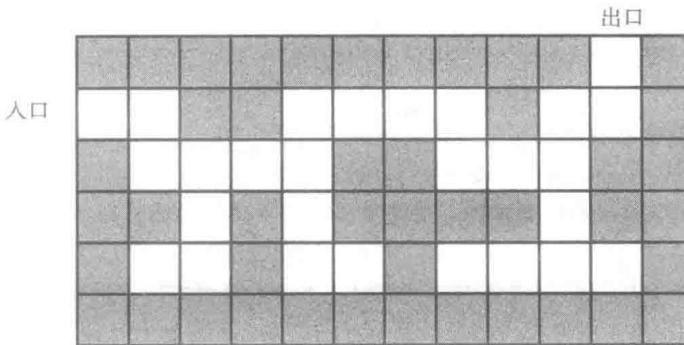


图 14-9 练习 12 所用的迷宫

## 项目

- 实现段 14.4 所给的 `solveTower` 算法的两个版本。可以用字符串或是单个字符表示塔。每个方法应该显示一些说明，用来表示必须进行的盘子移动。在每个方法内插入计数器，用来统计调用的次数。这些计数器可以作为这些方法所在类的数据域。对于不同的盘子数，比较每个方法递归调用的次数。
- 使用下列迭代算法可以求解汉诺塔问题。从柱 1 开始，按柱 1、柱 3、柱 2、柱 1 等的次序在柱间移动盘子，根据下列规则对每根柱子最多移动一步：
  - 将最上面的盘子按特定的次序从一个柱子移到下一个可能的柱子。记住，你不能将盘子放在较小的盘子上面。
  - 如果你准备要移动的盘子是所有盘子里的最小盘子，且你刚刚将它移到当前的柱子上，则不需要移动它。而是考虑下一个盘子。
 这个算法与段 14.4 给出的递归算法及图 14-2 所示的图有相同的移动过程。所以这个迭代算法也是  $O(2^n)$  的。
- 实现这个算法。
- 写一个应用或小应用程序，给出汉诺塔问题求解的动画。要求你将  $n$  个盘子从一个柱子移到另一个上，一次移动一个。你只能移动柱子最上面的一个盘子，只能将盘子放到柱子上更大的盘子上面。因为每个盘子都有特定的特性，比如它的大小，自然要为盘子定义一个类。
 

设计并实现 ADT 塔，包括下列操作：

  - 将一个盘子添加到柱子上盘子的最上面
  - 移走最上面的盘子
 还要设计并实现解决这个问题的递归方法所在的类。
- 设计并实现前缀表达式的类。包括识别前缀表达式的方法及求值的另一个方法。写程序论证你的类。
- 考虑你在练习 5 中为后缀表达式语言所写的语法。
  - 为后缀表达式的识别算法写伪代码。
  - 设计并实现后缀表达式的类。包括识别后缀表达式的方法及求值的另一个方法。写程序论证你的类。
- 重做项目 5，但在计算后缀表达式的值的方法中使用递归。
- 下面是一个语法，允许你当优先级规则能消除歧义时忽略中缀表达式中的括号。例如， $a+b*c$  表示的是  $a+(b*c)$ 。但是当会产生歧义时语法需要括号。也就是说，当几个运算符具有相同的优先级时，语法不允许从左到右的结合律。例如， $a/b*c$  是非法的。注意，定义中引入了因子和项：
 
$$<\text{表达式}> = <\text{项}> | <\text{项}> + <\text{项}> | <\text{项}> - <\text{项}>$$

<项>=<因子>|<因子>\*<因子>|<因子>/<因子>  
 <因子>=<字母>|(<表达式>)  
 <字母>=a|b|…|z

设计并实现所给语法描述的中缀表达式的类。包括识别合法中缀表达式的方法。识别算法基于子任务的递归链：

找到一个表达式 -> 找到一个项 -> 找到一个因子

因为找到一个表达式用到找到一个项，后者又用到找到一个因子，所以这是一个递归链。找到一个因子检测到一个基础情形，或是用到了找到一个表达式，所以形成递归链。这个识别算法的伪代码如下：

```

// The grammar specifies that an expression is either a single term or
// a term followed by a + or a -, which then must be followed by a second term.
findAnExpression()
{
 findATerm()
 if(下一个符号是+或-)
 findATerm()
}

// The grammar specifies that a term is either a single factor or
// a factor followed by a * or a /, which must then be followed by a second factor.
findATerm()
{
 findAFactor()
 if(下一个符号是*或/)
 findAFactor()
}

// The grammar specifies that a factor is either a single letter (the base case) or
// an expression enclosed in parentheses.
findAFactor()
{
 if(第一个符号是一个字母)
 完成
 else if(第一个符号是'(')
 {
 找到'('后面字符开始的一个表达式
 检查')'
 }
 else
 不存在因子
}

```

8. 假定有一排  $n$  个灯，在特定条件下可以点亮或灭掉，规则如下。第一个灯可以在任何时间亮或灭。其他灯中的任一个，仅当前一个灯是亮时，或是它之前的所有灯都灭时，可以亮或灭。如果初始时所有灯都是亮的，你如何能使它们都灭掉？对于编号为 1 ~ 3 的 3 个灯，你可以采取下列步骤，其中 1 表示灯亮，0 表示灯灭：

1 1 1 初始时全部都亮

0 1 1 关掉 1 号灯

0 1 0 关掉 3 号灯

1 1 0 打开 1 号灯

1 0 0 关掉 2 号灯

0 0 0 关掉 1 号灯

a. 写伪代码定义下列相互递归方法，用来求解本问题：

```

// Turns off n lights that are initially on.
turnOff(n)

```

```
// Turns on n lights that are initially off.
turnOn(n)
```

- b. 写一个程序，显示关掉初始全亮的  $n$  个灯的步骤，或是打开初始全灭的  $n$  个灯的步骤。  
c. 在解决有  $n$  个灯的问题中，表示灯在亮或灭间切换的次数的递推关系是什么？  
9. 设计并实现一个算法，使用递归和回溯将整数数组按升序排序。考虑给定的数组作为输入，得到的有序数组作为输出。每次你从输入数组中拿到一个整数，将它放到输出数组的结尾。如果结果是无序的，则回溯。  
10. 使用段 14.29 给出的递归算法 `searchFrom`，求解第 5 章项目 10 描述的迷宫问题。  
11. 实现一个算法，使用递归和回溯求解  $n$  皇后问题。使用你的程序找到 8 皇后问题的所有解。  
12. 小波是数学函数，可用来在压缩前转换信号、图像和视频。最简单的小波函数之一是 Haar 变换。它递归地使用平均值和差值并使用以下公式处理信号：

$$\text{信号平均值} = (a + b) / 2, \text{ 其中 } a \text{ 和 } b \text{ 是两个相邻的信号值或像素}$$

$$\text{差值} = b - a$$

例如，我们将 Haar 变换应用于图 14-10a 所示的原始信号值的一维数组。我们先处理整个数组，比较每一对项并找到平均值和差值。图 14-10b 展示了这些计算结果的二分之一分辨率信号。注意到可以将平均值和差值直接保存到原始数组中，或者可以使用一个临时数组，然后将其复制到原始数组。

我们递归地重复处理平均值对，得到新的平均值和差值，从而得到如图 14-10c 和图 14-10d 所示的四分之一分辨率信号和八分之一分辨率信号。在这个图的图 14-10d 中，注意到有一个平均值。这是递归的基础情形。

此时，重建数组如下。最终的平均值是数组中的第一项。然后从最低分辨率级（本例中是八分之一）开始，将差值追加到数组中。变换信号如图 14-10e 所示。通过将低于某个阈值的值设置为 0 来压缩结果。

实现用于一维信号，例如 AIFF 音频文件的 Haar 变换。

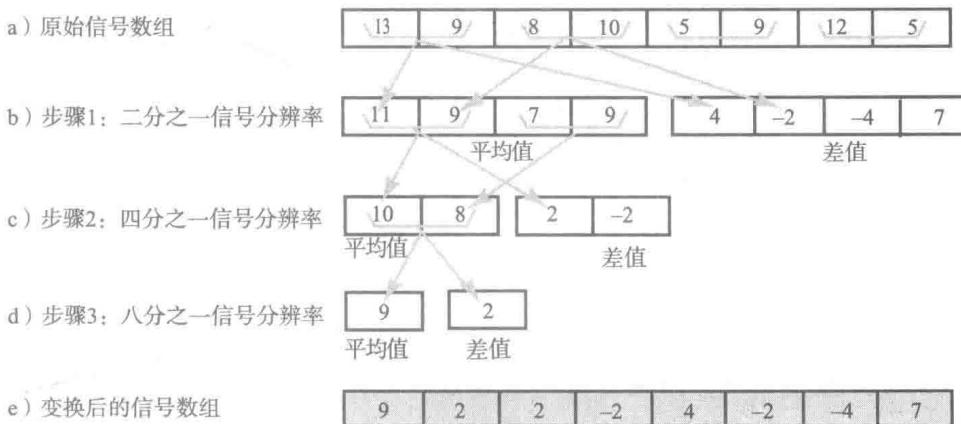


图 14-10 项目 12 中信号变换示例

## Java 插曲 5 |

Data Structures and Abstractions with Java, Fifth Edition

# 再论泛型

先修章节：Java 插曲 1

本插曲继续 Java 插曲 1 中对泛型和接口的讨论。在接下来的两章中将使用 Java 的这些内容，来介绍将对象按升序或降序排列的方法。

## 接口 Comparable

J5.1 方法 `compareTo`。补充材料 1 (在线) 描述了用于类 `String` 的方法 `compareTo`。这个方法返回一个整数，作为两个字符串的比较结果。例如，如果 `s` 和 `t` 都是字符串，则 `s.compareTo(t)` 的结果是

- 负数，如果 `s` 在 `t` 的前面
- 零，如果 `s` 和 `t` 相等
- 正数，如果 `s` 在 `t` 的后面

其他的类可以有自己的 `compareTo` 方法，其行为类似。

 注：方法 `compareTo` 比较两个对象，并返回一个表示比较结果的带符号整数。例如，如果 `x` 和 `y` 是实现了接口 `Comparable` 的同一个类的两个实例，则 `x.compareTo(y)` 返回

- 一个负数，如果 `x` 小于 `y`
- 零，如果 `x` 等于 `y`
- 一个正数，如果 `x` 大于 `y`

如果 `x` 和 `y` 有不同的类型，则 `x.compareTo(y)` 会抛出 `ClassCastException` 异常。

 注：枚举有 `compareTo` 方法。枚举时枚举对象出现的次序决定比较的结果。例如，如果有

```
enum Coin {PENNY, NICKEL, DIME, QUARTER}
```

且 `myCoin` 是 `Coin` 的实例，则 `myCoin.compareTo(Coin.DIME)` 能判定 `myCoin` 是位于 `Coin.DIME` 的前面、后面或是两者相等。具体来说，如果 `myCoin` 的值是 `Coin.PENNY`，则与 `Coin.DIME` 比较的结果将得到一个负整数。

J5.2 定义了方法 `compareTo` 的所有类，实现了 Java 类库中包 `java.lang` 中的标准接口 `Comparable`。显示在程序清单 JI5-1 中的这个接口，使用泛型 `T` 表示实现接口的类。所以，通过调用 `compareTo` 方法，可以比较类 `T` 的两个对象。

### 程序清单 JI5-1 接口 `java.lang.Comparable`

```
1 package java.lang;
2 public interface Comparable<T>
3 {
```

```

4 public int compareTo(T other);
5 } // end Comparable

```

现在来创建类 Circle，让其具有方法 equals 和 compareTo，及序言中程序清单 P-1 J5.3 中给出的接口 Measurable 中的方法。这个类实现了两个接口，类的开头如下所示。

```

public class Circle implements Comparable<Circle>, Measurable
{
 private double radius;
 <构造方法和其他方法的定义>

```

类名出现在接口名 Comparable 后的尖括号中。所以 Circle 对应于接口中的 T，它是 compareTo 参数的数据类型。

类中方法 compareTo 的实现如下：

```

public int compareTo(Circle other)
{
 int result;
 if (this.equals(other))
 result = 0;
 else if (radius < other.radius)
 result = -1;
 else
 result = 1;
 return result;
} // end compareTo

```

上述 compareTo 方法通过比较圆的半径对圆进行比较，且假定 Circle 有自己的 equals 方法。虽然 compareTo 不需要调用 equals，但这两个方法通常应返回一致的结果。即如果 object1.equals(object2) 为真，则 object1.compareTo(object2) 应该返回零。

虽然上面这个版本的 compareTo 方法，对不相等的对象返回 -1 或是 +1，但 compareTo 的规范说明中并没有坚持必须是这些值。只要求结果的符号必须是正确的。所以，当对整数进行比较时，简单减法常能得到合适的返回值。例如，如果类 Circle 中的数据域 radius 是整数而不是实数，compareTo 可以有如下这般的定义：

```

// Assumes radius is an integer
public int compareTo(Circle other)
{
 return radius - other.radius;
} // end compareTo

```

你或许会奇怪，compareTo 为什么不属于类 Object。原因是，不是所有的类都应该有一个 compareTo 方法。有的对象类就可能没有自然序，这也是司空见惯的事情。例如，考虑邮寄地址的类。判定两个地址是否相等应该很简单，但一个地址小于另一个地址是什么意思？



注：并非所有的类都应该实现接口 Comparable。



学习问题 1 定义一个类 Name，实现接口 Comparable 和序言中程序清单 P-2 给出的接口 NameInterface。

## 泛型方法

J5.6 假定有一个类，在它的头部没有定义类型参数，但在这个类的方法中想使用泛型数据类型。例如，可能有一个能执行不同实用功能的静态方法的类。Java 类库中的 `Math` 类就是这样的一个类。可以采用下列步骤来写一个这样的泛型方法（generic method）：

- 在尖括号中写上类型参数，放在方法头部返回类型的前面。
- 在方法内使用类型参数，如同它在泛型类中你的处理那样，即或作为返回类型、方法参数的数据类型，或作为方法体内变量的数据类型。

程序清单 JI5-2 中给出泛型方法 `displayArray` 示例，它显示有泛型类型项的数组的内容。`main` 方法调用 `displayArray`，先是传给它一个字符串数组，然后再传给它一个字符数组。

**程序清单 JI5-2 泛型方法示例**

```

1 public class Example
2 {
3 public static <T> void displayArray(T[] anArray)
4 {
5 for (T arrayEntry : anArray)
6 {
7 System.out.print(arrayEntry);
8 System.out.print(' ');
9 } // end for
10 System.out.println();
11 } // end displayArray
12
13 public static void main(String args[])
14 {
15 String[] stringArray = {"apple", "banana", "carrot", "dandelion"};
16 System.out.print("stringArray contains ");
17 displayArray(stringArray);
18
19 Character[] characterArray = {'a', 'b', 'c', 'd'};
20 System.out.print("characterArray contains ");
21 displayArray(characterArray);
22 } // end main
23 } // end Example

```

输出

```

stringArray contains apple banana carrot dandelion
characterArray contains a b c d

```

**学习问题 2 定义泛型方法 swap，交换所给数组中两个指定位置的对象。**



## 限定的类型参数

J5.7 在前面的程序段中，泛型数据类型表示客户选择的任意的类类型。但有些情形下，需要你限制或控制客户的选择。例如，考虑下列简单的正方形类：

```

public class Square<T>
{
 private T side;

 public Square(T initialSide)

```

```

{
 side = initialSide;
} // end constructor

public T getSide()
{
 return side;
} // end getSide
} // end Square

```

通过如下的语句，可以创建一个 `Square<Integer>` 对象和一个 `Square<Double>` 对象：

```

Square<Integer> intSquare = new Square<>(5);
Square<Double> realSquare = new Square<>(2.1);

```

还可以创建边不是数值的正方形：

```

Square<String> stringSquare = new Square<>("25");

```

但这个灵活性是个问题。

让类 `Square` 包含一个返回其面积的方法，如下所示。 J5.8

```

public double getArea()
{
 double s = side.doubleValue();
 return s * s;
} // end getArea

```

编译程序将给出下列错误信息：

```

error: cannot find symbol
double s = side.doubleValue();
^
symbol: method doubleValue()
location: variable side of type T
where T is a type-variable:
T extends Object declared in class Square

```

因为 `T` 表示任意的类类型，且所有的类都派生于 `Object`，故编译程序不能辨别 `side` 有没有 `doubleValue` 方法。例如，如果 `side` 指向一个字符串，它就没有这个方法。

我们想让正方形的边是一个数值量。让 `T` 表示派生于 `Number` 的类来施加这个限制，J5.9  
`Number` 类是类 `Byte`、`Double`、`Float`、`Integer`、`Long` 和 `Short` 的基类（超类）。所以，在 `Square` 的头部写 `T extends Number`，就限定（bound）了 `T`：

```

public class Square<T extends Number>

```

这里，`Number` 可称为 `T` 的上限（upper bound）。现在 `Square` 可以含有前面提到的 `getArea` 方法了。另外，编译程序会拒绝创建其边是字符串的任何尝试。例如，语句

```

Square<String> stringSquare = new Square<>("25");

```

会让编译程序发出下列信息：

```

error: type argument String is not within bounds of type-variable T
Square<String> stringSquare = new Square<>("25");
^
where T is a type-variable:
T extends Number declared in class Square

```

 J5.10 示例。假定想写一个静态方法，它返回数组中的最小对象。数组中的对象可以是字符串、`Integer` 对象或是可比较的任何对象。即这些对象的类必须实现了接口

`Comparable`。我们需要客户提供要比较的对象数组。

假定方法的实现如下所示。

```
public MyClass
{
 // First draft and INCORRECT:
 public static <T> T arrayMinimum(T[] anArray)
 {
 T minimum = anArray[0];
 for (T arrayEntry : anArray)
 {
 if (arrayEntry.compareTo(minimum) < 0)
 minimum = arrayEntry;
 } // end for

 return minimum;
 } // end arrayMinimum
 .
}
```

因为泛型 `T` 可以表示任何的类类型，故客户可以向这个方法传递一个没有 `compareTo` 方法的对象数组。因为这个原因，编译程序将发出语法错误信息。

要让这个方法正确，就必须限定 `T`，以便让它表示提供了方法 `compareTo` 的类类型。为此，将方法头部中的 `<T>` 替换为 `<T extends Comparable<T>>`。所以头部变为

```
public static <T extends Comparable<T>> T arrayMinimum(T[] anArray) // CORRECT
```

如果类 `Gadget` 实现了 `Comparable`，且如果 `myArray` 是 `Gadget` 对象的数组，则客户可以如下调用 `arrayMinimum`：

```
Gadget smallestGadget = MyClass.arrayMinimum(myArray);
```

编译程序会发现 `myArray` 含有 `Gadget` 对象。不过，如果 `myArray` 含有没有实现 `Comparable` 的类对象时，编译程序会报错。

### 注：可以限定类型参数的类型

任何类、接口或是枚举类型——即使是参数——都可以限定类型参数。基本类型及数组类型不能被限定。

 安全说明：使用泛型数据类型替代 `Object`，提供了让编译程序检查无效数据类型的一种方法，提高了代码的安全度。



### 学习问题 3 下面的类有错误吗？如果有，是什么错误？

```
public final class Min
{
 public static T smallerOf(T x, T y)
 {
 if (x < y)
 return x;
 else
 return y;
 } // end smallerOf
} // end Min
```

## 通配符

问号 ? 用来表示一个未知的类类型，称为通配符 (wildcard)。要说明它的含义，先来看看 Java 插曲 1 中，程序清单 JI1-2 所给的 OrderedPair 类的客户程序里面的几条语句。首先，用下面的语句创建 OrderedPair<?> 类型的一个变量：

```
OrderedPair<?> aPair;
```

现在可以给这个变量赋值一对字符串，如下所示

```
aPair = new OrderedPair<>("apple", "banana"); // A pair of String objects
```

或是赋值一对 Integer 对象，如下所示

```
aPair = new OrderedPair<>(1, 2); // A pair of Integer objects
```

或是赋值任何其他类型的一对对象。

现在来看静态方法

```
public static void displayPair(OrderedPair<?> pair)
{
 System.out.println(pair);
} // end displayPair
```

及下列对象：

```
OrderedPair<String> aPair = new OrderedPair<>("apple", "banana");
OrderedPair<Integer> anotherPair = new OrderedPair<>(1, 2);
```

方法 displayPair 将接收一对对象作为参数，其数据类型是任意的类，如下列语句所示：

```
displayPair(aPair);
displayPair(anotherPair);
```

但是，如果在方法中用 Object 替换通配符，方法的头如下

```
public static void displayPair(OrderedPair<Object> pair)
```

则前面对 displayPair 的两个调用都是非法的，会导致编译程序弹出下列信息：

```
error: incompatible types: OrderedPair<String> cannot be converted to
OrderedPair<Object>
error: incompatible types: OrderedPair<Integer> cannot be converted to
OrderedPair<Object>
```

## 限定的通配符

回忆段 J5.10 中所讨论的方法 arrayMinimum：

```
public MyClass
{
 public static <T extends Comparable<T>> T arrayMinimum(T[] anArray)
 {
 T minimum = anArray[0];
 for (T arrayEntry : anArray)
 {
 if (arrayEntry.compareTo(minimum) < 0)
 minimum = arrayEntry;
 }
 }
}
```

```

 } // end for

 return minimum;
} // end arrayMinimum

...

```

用下列语句调用这个方法

```
Gadget smallestGadget = MyClass.arrayMinimum(myArray);
```

其中，`myArray` 是 `Gadget` 对象的数组。因为 `arrayMinimum` 定义中的表达式 `T extends Comparable<T>` 限定了泛型数据类型 `T`，所以 `Gadget` 必须实现接口 `Comparable<Gadget>`。但坚持让 `Gadget` 对象仅能与另一个 `Gadget` 对象进行比较，这又限制得过多了。如果我们从 `Widget` 派生 `Gadget`，其中 `Widget` 实现了 `Comparable<Widget>`，如图 J15-1 所示，又会怎样呢？如果 `gadget`（工具）和 `widget`（部件）非常相似，有相同的比较基准，则 `Gadget` 应该使用从 `Widget` 继承来的方法 `compareTo`，而不是再定义自己的。但另一方面，将含 `gadget` 和 `widget` 的数组作为参数来调用方法 `arrayMinimum` 时，又不能编译。

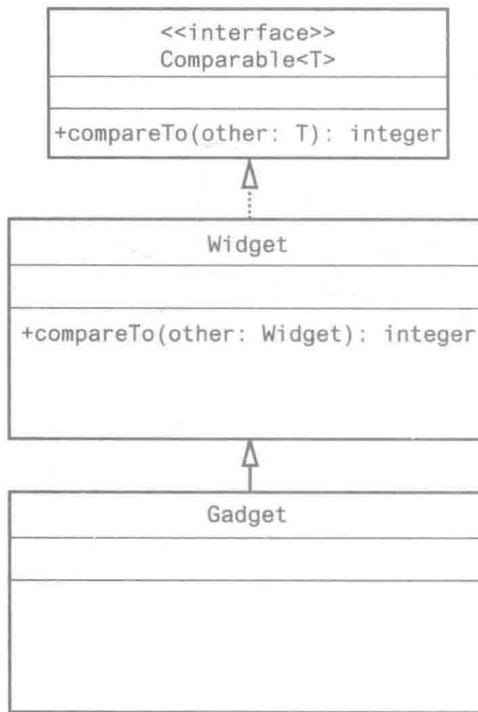


图 J15-1 类 `Gadget` 派生于类 `Widget`，后者实现了接口 `Comparable`

不是让 `T` 的对象仅能与 `T` 的另一个对象进行比较，而是允许与 `T` 的超类的对象进行比较。这样，不是写 `T extends Comparable<T>`，而是写

```
T extends Comparable<? super T>
```

通配符 `?` 表示任意的类类型，但符号 `? super T` 表示 `T` 的任意超类。回忆一下，我们在第 7 章程序清单 7-5 中出现的接口 `PriorityQueueInterface` 中用过这个记号。所以，方法 `arrayMinimum` 的头部应该是

```
public static <T extends Comparable<? super T>> void arrayMinimum(T[] a)
```

**!** 程序设计技巧：要让 Comparable 应用于任意类型，可写 Comparable<? super T> 替代 Comparable<T>。

**♪** **注：限定的通配符**

当使用泛型时，通配符 ? 表示任意类。限定或限制通配符的方法有两种。例如，? super Gizmo 表示 Gizmo 的任意超类。我们说 Gizmo 是通配符的下限 (lower bound)。类似地，? extends Gizmo 表示 Gizmo 的任意子类。其中 Gizmo 是通配符的上限 (upper bound)。在第 4 章段 4.13 和段 4.17 中给出了术语“上限”和“下限”的其他含义。

**♪** **注：泛型类和接口**

定义泛型数据类型的任意类或接口，可以使用本插曲从段 J5.9 开始描述的符号来限定类型。

# 排序简介

先修章节：第 3 章、第 4 章、第 9 章、Java 插曲 5

## 目标

学习完本章后，应该能够

- 使用下列方法将一个数组按升序排序：选择排序、插入排序和希尔排序
- 使用插入排序将结点链表按升序排序
- 评估排序的效率，讨论不同方法的相对效率

我们都很熟悉将对象按从最小到最大或从最大到最小的次序排列。我们不仅是对数值这样处理，我们还按身高、年龄或是名字对人进行排序；按歌名、作者或是唱片对乐曲进行排序；等等。将事物按升序或降序重排称为 **排序**（sorting）。可以对能互相进行比较的任意项的集合进行排序。如何精确地比较两个对象，依赖于对象的属性。例如，可以对书架上的一排书按几种不同方式重排：按书名、按作者、按书高、按颜色等。当实现方法 `compareTo` 时，书这个对象所属类的设计者应该选择这其中的一种。

假定你有项的一个集合，必须以某种方式进行排序。例如，你或许想将一组数从最小到最大或从最大到最小重排，或者想将一些字符串按字母序重排。本章讨论并实现一些简单的算法，将项按升序排序。即我们的算法将重排集合中的前  $n$  项，使其满足

$$\text{项 1} \leqslant \text{项 2} \leqslant \dots \leqslant \text{项 } n$$

对算法进行少许的修改，就能实现项的降序排序。

对数组进行排序，通常比对结点链表排序要简单些。因此，一般的排序算法是对数组排序。特别地，我们的算法对数组  $a$  中的前  $n$  个值进行重排，以得到

$$a[0] \leqslant a[1] \leqslant a[2] \leqslant \dots \leqslant a[n - 1]$$

不过，我们还将使用其中的一个算法对结点链表进行排序。

排序是这样一个常见且重要的任务，因此已有很多排序算法问世。本章介绍用于排序数据的这些基本算法。虽然大多数例子是对整数进行排序，但用 Java 所实现的这些算法可以对任意的 `Comparable` 对象——即实现了接口 `Comparable` 的任意类的对象，故定义了方法 `compareTo`——进行排序。

排序算法的效率很重要，特别是当涉及的数据量很大时。我们将考查本章中各算法的性能，会发现它们相对较慢。下一章将提出通常会快得多的排序算法。

## 对数组进行排序的 Java 方法的组织

15.1

对数组进行排序的方法的一种组织方式是创建一个实现不同排序的静态方法的类。方法中，为数组中的对象定义泛型  $T$ 。例如，我们可以将这样的一个方法的方法头表示如下：

```
public static <T> void sort(T[] a, int n)
```

这里，数组  $a$  可以含有任意类的对象， $n$  是待排序数组中项的个数。

对于要排序的数组，数组中的对象必须是 Comparable 的。所以 T 表示的类必须实现接口 Comparable。为确保这一要求，我们在排序方法的方法头中，在返回类型的前面，用

```
<T extends Comparable<T>>
```

来替代简单的 <T>。然后使用 T 作为参数及方法内局部变量的数据类型。例如，类的开头可以是这样的：

```
public class SortArray
{
 public static <T extends Comparable<T>> void sort(T[] a, int n)
 { . . . }
```

但是，正如前一个 Java 插曲的最后提到的，如下的语句，能让我们对 T 的超类的对象进行比较：

```
T extends Comparable<? super T>
```

所以，方法 sort 的方法头应该是这样的：

```
public static <T extends Comparable<? super T>> void sort(T[] a, int n)
```

现在集中关注对数组进行排序的几种方法。

## 选择排序

假定你想将书架上的书按高度重排，最矮的书在最左边。你或许先将所有的书都扔到地上。然后再把它们一本本地按照合适的次序拾到书架上。如果先拿回最矮的书放到书架上，然后是下一本最矮的书，等等，则你完成的是一种选择排序（selection sort）。但使用地板——或另一个书架——来临时存放这些书，用到了不必要的额外空间。

换一种做法，走到原来的书架前，选择最矮的书。因为你想将它放到书架的第一本，所以拿走书架上的第一本，将最矮的书放到这个位置。此时你手上仍然有一本书，故你将它放到刚才最矮的书所在的地方。也就是说，最矮的书与第一本书交换了位置，如图 15-1 所示。现在，忽略最矮的书，对书架上的其余的书重复这个过程。

就数组 a 来说，选择排序找到数组中最小项，将它与 a[0] 相交换。然后，忽略 a[0]，排序找到下一个最小的项并交换到 a[1]，依此类推。注意到，我们仅使用一个数组。通过将项与其他项进行交换来排序。

可以将数组复制到第二个数组中，然后将项复制回原来的数组中的适当位置。但是，这与使用地板暂时放书是一样的。所幸的是，这个额外的空间都不是必需的。

图 15-2 显示了选择排序是如何将整数数组通过交换值完成的排序。从原始数组开始，排序找到数组中的最小值，即 a[3] 中的 2。a[3] 中的值与 a[0] 中的值相交换。交换后，最小值位于 a[0] 中，这也是它应的地方。

下一个最小值是 a[4] 中的 5。然后排序方法交换 a[4] 中的值与 a[1] 中的值。到此，a[0] 及

15.2

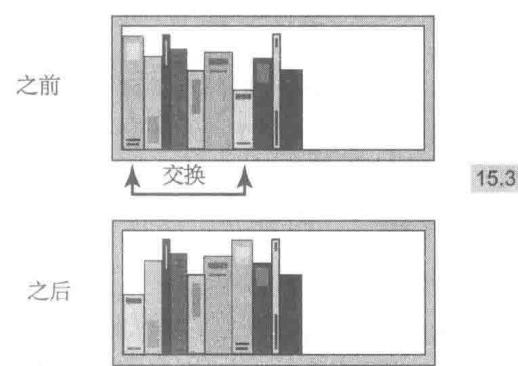


图 15-1 将最矮的书与第一本书交换之前及之后

$a[1]$  中的值是数组中的最小值，且已在最终有序数组的正确位置上。接下来，算法交换下一个最小值——8——与  $a[2]$ ，以此类推，直到整个数组有序时为止。

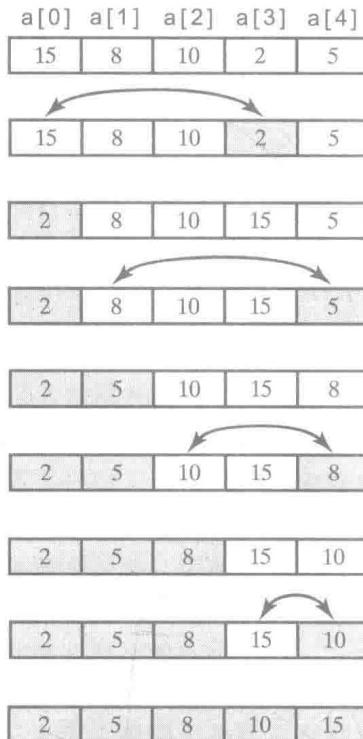


图 15-2 使用选择排序将整数数组排成升序

## 迭代的选择排序

### 15.4

下列伪代码描述迭代的选择排序算法：

```

Algorithm selectionSort(a , n)
// Sorts the first n entries of an array a .
for ($index = 0$; $index < n - 1$; $index++$)
{
 $indexOfNextSmallest = a[index], a[index + 1], \dots, a[n - 1]$ 中最小值的下标
 交换 $a[index]$ 和 $a[indexOfNextSmallest]$ 的值
 // Assertion: $a[0] \leq a[1] \leq \dots \leq a[index]$, and these are the smallest
 // of the original array entries. The remaining array entries begin at $a[index + 1]$.
}

```

注意到，在 **for** 循环的最后一次迭代中， $index$  的值是  $n-2$ ，虽然数组的最后一项在  $a[n-1]$  中。一旦  $a[0]$  到  $a[n-2]$  中的各项都已在它们正确的位罝了，就只需要放置剩下的最后一项  $a[n-1]$  了。但因为其他的项都已被正确放置，那这最后一项也一定在其正确的位罝。



### 注：记号

在数学中，使用单字母命名变量是常见的。认识到这一点，并设法节省一些空间，我们在正文及伪代码中使用  $a$  和  $n$  分别表示数组和它的项数。但在 Java 代码中，我们试图避开单字母标识符，只将它们用作数组下标或循环变量。但是，本章和下一章看到的代码中使用  $a$  和  $n$  只为了与正文保持一致。

程序清单 15-1 中的类含有公有方法 `selectionSort` 和辅助排序的两个私有方法。当我们开发了其他的排序方法后，可以再添加进来。

容易看到，`selectionSort` 的定义是将前一个伪代码直接翻译为 Java 代码。方法 `getIndexOfSmallest` 查找数组中从 `a[first]` 到 `a[last]` 的各项，并返回其中最小项的下标。方法中用到了两个局部变量 `min` 和 `indexOfMin`。查找过程中，`min` 指向到目前为止找到的最小值。这个值位于 `a[indexOfMin]` 中。查找结束时，方法返回 `indexOfMin`。注意这里我们的意图，我们本可以假定 `last` 总是 `n-1`，从而不让它出现在参数中。不过，现在这个通用版本还能用在其他的设置下。

因为交换数组中的项时没有调用方法 `compareTo`，所以方法 `swap` 可以简单地用 `Object` 作为这些项的类型。

### 程序清单 15-1 使用选择排序对数组进行排序的类

```

1 /**
2 * Class for sorting an array of Comparable objects from smallest to largest.
3 */
4 public class SortArray
5 {
6 /** Sorts the first n objects in an array into ascending order.
7 * @param a An array of Comparable objects.
8 * @param n An integer > 0. */
9 public static <T extends Comparable<? super T>>
10 void selectionSort(T[] a, int n)
11 {
12 for (int index = 0; index < n - 1; index++)
13 {
14 int indexOfNextSmallest = getIndexOfSmallest(a, index, n - 1);
15 swap(a, index, indexOfNextSmallest);
16 // Assertion: a[0] <= a[1] <= . . . <= a[index] <= all other a[i].
17 } // end for
18 } // end selectionSort
19
20 // Finds the index of the smallest value in a portion of an array a.
21 // Precondition: a.length > last >= first >= 0.
22 // Returns the index of the smallest value among
23 // a[first], a[first + 1], . . . , a[last].
24 private static <T extends Comparable<? super T>>
25 int getIndexOfSmallest(T[] a, int first, int last)
26 {
27 T min = a[first];
28 int indexOfMin = first;
29 for (int index = first + 1; index <= last; index++)
30 {
31 if (a[index].compareTo(min) < 0)
32 {
33 min = a[index];
34 indexOfMin = index;
35 } // end if
36 // Assertion: min is the smallest of a[first] through a[index].
37 } // end for
38
39 return indexOfMin;
40 } // end getIndexOfSmallest
41
42 // Swaps the array entries a[i] and a[j].
43 private static void swap(Object[] a, int i, int j)
44 {
45 Object temp = a[i];

```

```

46 a[i] = a[j];
47 a[j] = temp;
48 } // end swap
49 } // end SortArray

```

学习问题 1 使用选择排序把数组 9 6 2 4 8 按升序排序时，跟踪排序步骤。



## 递归的选择排序

**15.6** 选择排序也有自然的递归方式。涉及数组的递归算法常常对数组的一部分进行操作。这样的算法使用两个形参 `first` 和 `last`，标出从项 `a[first]` 到项 `a[last]` 的数组部分。程序清单 15-1 中的方法 `getIndexOfSmallest` 说明了这个技术。递归的选择排序算法使用的也是这个名字：

```

Algorithm selectionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
 indexOfNextSmallest = a[first], a[first + 1], . . . , a[last] 中最小值的下标
 交换 a[first] 和 a[indexOfNextSmallest] 的值
 // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[first] and these are the smallest
 // of the original array entries. The remaining array entries begin at a[first + 1].
 selectionSort(a, first + 1, last)
}

```

将最小项放到数组的第一个位置后，忽略它并使用选择排序对数组的其他部分进行排序。如果数组只有一个项，则不需要排序。这种情形下，`first` 和 `last` 是相等的，所以算法保持数组不改变。

**15.7** 当使用 Java 语言实现前面这个递归算法时，得到的方法的形参包括了 `first` 和 `last`。所以，它的方法头将与段 15.5 给出的迭代方法 `selectionSort` 的方法头不一样。不过我们提供下列方法来调用递归方法：

```

public static <T extends Comparable<? super T>
 void selectionSort(T[] a, int n)
{
 selectionSort(a, 0, n - 1); // Invoke recursive method
} // end selectionSort

```

你可以决定让递归方法 `selectionSort` 是私有的还是公有的，而如果让它成为公有的，则可以为用户提供两种调用排序方法的选择。类似地，可以使用参数 `first` 和 `last` 修 改段 15.5 节给出的迭代选择排序，(见练习 6)，然后提供一个调用它的方法。

记着这些结论，我们让后面的排序算法有 3 个形参——`a`、`first` 和 `last`——使得它们更通用，能对 `a[first]` 到 `a[last]` 间的项进行排序。

## 选择排序的效率

**15.8** 在迭代方法 `selectionSort` 中，`for` 循环执行  $n-1$  次，所以它分别调用 `getIndexOfSmallest` 和 `swap` 方法各  $n-1$  次。在  $n-1$  次调用 `getIndexOfSmallest` 中，`last` 是  $n-1$ ，而 `first` 从 0 变到  $n-2$ 。每次调用 `getIndexOfSmallest` 时，它的循环要执行 `last-`

`first` 次。因为 `last-first` 从  $(n-1)-0$  (即  $n-1$ ) 变到  $(n-1)-(n-2)$  (即 1)，故这个循环总共执行了

$$(n-1) + (n-2) + \dots + 1$$

次。这个和是  $n(n-1)/2$ 。所以，因为循环中的每个操作都是  $O(1)$  的，故选择排序是  $O(n^2)$  的。注意到，我们的讨论不依赖于数组中数据的初始情况。它可以是完全无序的、接近有序的或是完全有序的；任何情形下，选择排序都是  $O(n^2)$  的。

递归的选择排序与迭代的选择排序执行相同的操作，所以它也是  $O(n^2)$  的。

### 注：选择排序的时间效率

选择排序是  $O(n^2)$  的，不论数组中项的初始次序如何。虽然排序需要  $O(n^2)$  次比较，但它仅执行  $O(n)$  次交换。所以选择排序仅有很少的数据移动。

## 插入排序

另一个直观的排序算法是插入排序 (insertion sort)。还是假定你想对书架上的书按高度重排，最矮的书在最左边。如果书架上最左边的书是仅有的一本书，则你的书架已是有序的了。另一种情况，你还有其他的书要排序。考虑第二本书。如果它比第一本书高，现在你有两本有序的书了。如果不是，你拿走第二本书，将第一本书往右移，然后将你刚刚拿走的书插入书架上的第一个位置。现在前两本书已有序了。

现在考虑第三本书。如果它比第二本书高，则现在你已有 3 本有序的书了。否则，拿走第三本书，将第二本书往右移，如图 15-3a ~ 图 15-3c 所示。现在来看手中的书是否比第一本书高。如果是，将书插入书架上的第二个位置，如图 15-3d 所示。如果不是，将第一本书右移，将手中的书插入书架上的第一个位置。如果对剩余的每本书都重复这个过程，则你的书架将按书的高度重排有序。

图 15-4 所示为若干步插入排序后的书架。书架上左边的书是有序的。从书架上拿走一本待排序的书，将有序的书右移，一次移动一本，直到你为手上的书找到正确的放置位置为止。然后将这本书插入新的有序位置。

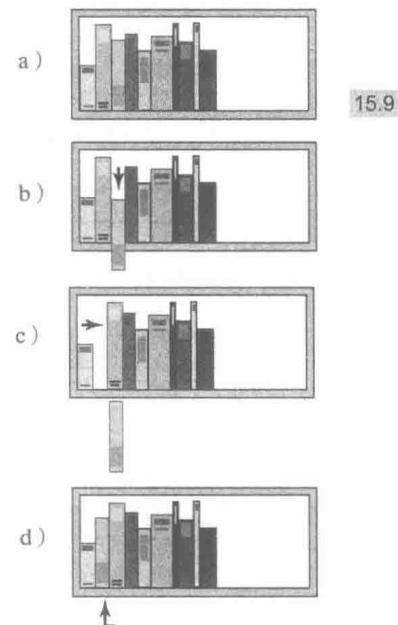


图 15-3 插入排序中第三本书的放置过程

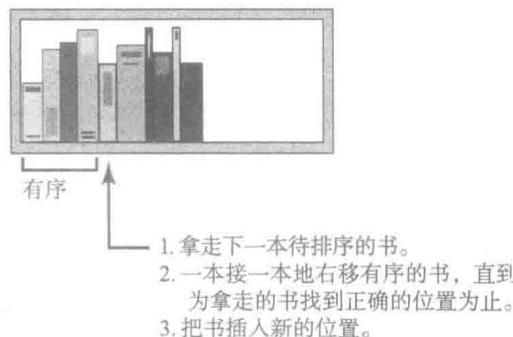


图 15-4 书的插入排序

## 迭代的插入排序

15.10

对数组的插入排序将数组划分 (partition) ——即分开——为两部分。第一部分是有序的，初始时仅含有数组中的第一项。第二部分含有其余的项。算法从未排序部分移走第一项，并将它插入有序部分中合适的有序位置。正如我们对书架所做的操作一样，从有序部分的末尾开始，持续朝着开头方向，通过将待排序项与各有序项进行比较来选择合适的位置。当比较时，移动有序部分的数组项，为插入腾出空间。

数组前 3 项已在正确位置的排序步骤如图 15-5 所示。3 是必须要放到有序区域中合适位置的下一项。因为 3 小于 8 和 5，但大于 2，所以移动 8 和 5，为 3 腾出位置。

图 15-6 所示是对整数数组完整的插入排序过程。算法的每一趟中，有序部分扩展一项，而未排序部分缩小一项。最后，未排序部分为空，数组有序。

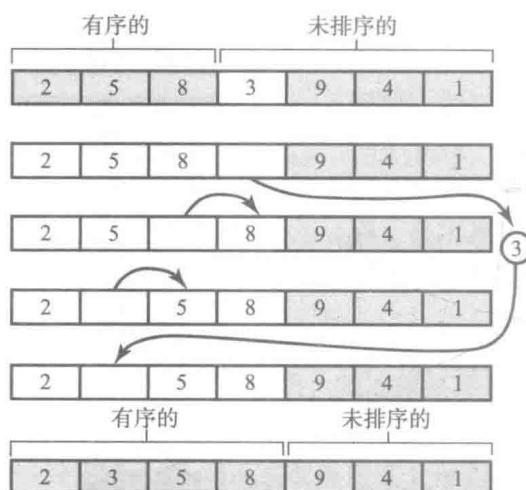


图 15-5 插入排序中，将下一个待排序项插入数组有序部分的合适位置

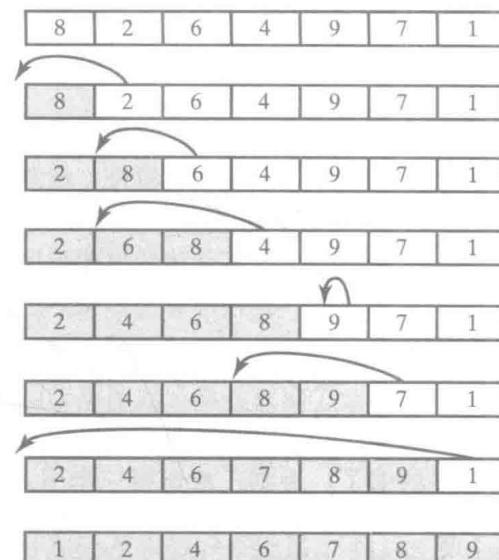


图 15-6 对整数数组进行升序排序的插入排序

下面的迭代算法描述了对数组 `a` 中从 `first` 到 `last` 之间的项进行的插入排序。要对数组中的前 `n` 项进行排序，对算法的调用应该是 `insertionSort(a, 0, n-1)`。

```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] iteratively.
for (unsorted = first + 1 through last)
{
 nextToInsert = a[unsorted]
 insertInOrder(nextToInsert, a, first, unsorted - 1)
}
```

有序部分含有一个项 `a[first]`，所以算法中的循环从下标 `first+1` 开始，处理未排序部分。然后调用另一个方法——`insertInOrder`——来执行插入。接在这个方法后面的伪代码中，`anEntry` 是要被插入到正确位置的值，`begin` 和 `end` 是数组的下标。

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted entries a[begin] through a[end].
index = end // Index of last entry in the sorted portion
// Make room, if needed, in sorted portion for another entry
```

```

while ((index >= begin) and (anEntry < a[index]))
{
 a[index + 1] = a[index] // Make room
 index--
}
// Assertion: a[index + 1] is available.
a[index + 1] = anEntry // Insert

```



学习问题 2 使用插入排序对数组 9 6 2 4 8 按升序排序，跟踪排序步骤。

## 递归的插入排序

插入排序可以递归地描述如下。如果对数组中除最后一个元素外的全部元素进行排序——比排序整个数组更小的问题——则可以将最后元素插入数组其他元素中的合适位置。15.11  
下列伪代码描述了递归的插入排序：

```

Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.
if (数组中含有1个以上的项)
{
 排序数组项a[first]到a[last - 1]
 将最后一项a[last]插入数组其余部分的正确有序位置
}

```

使用 Java 语言实现这个算法如下。

```

public static <T extends Comparable<? super T>>
 void insertionSort(T[] a, int first, int last)
{
 if (first < last)
 {
 // Sort all but the last entry
 insertionSort(a, first, last - 1);

 // Insert the last entry in sorted order
 insertInOrder(a[last], a, first, last - 1);
 } // end if
} // end insertionSort

```

**算法 insertInOrder：第一稿。**前一个方法可以调用之前给出的 *insertInOrder* 的迭代版本，或是这里描述的递归版本。如果要插入的项大于等于数组有序部分的最后一项，则这个插入项就放在最后项的后面，如图 15-7a 所示。否则，我们将有序部分的最后一项移到数组中下一个更高的位置，并将插入项插入剩余部分中，如图 15-7b 所示。15.12

可以更详细地描述这些步骤，如下所示。

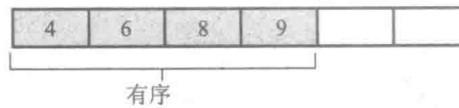
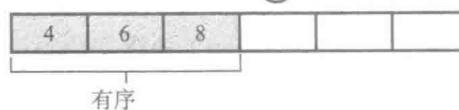
```

Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// First draft.

if (anEntry >= a[end])
 a[end + 1] = anEntry
else
{
 a[end + 1] = a[end]
 insertInOrder(anEntry, a, begin, end - 1)
}

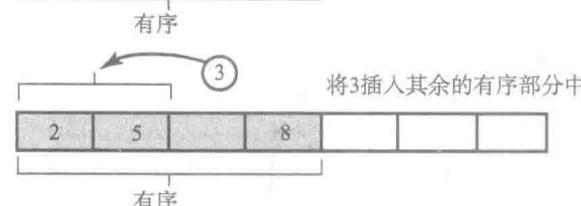
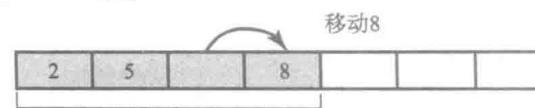
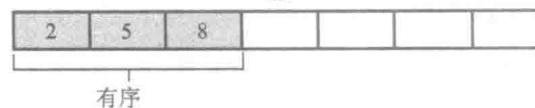
```

(9)  $9 > 8$ , 所以它应该放在8的后面



a) 项大于等于有序部分的最后一项

(3)  $3 < 8$ , 所以……



有序

b) 项小于有序部分的最后一项

图 15-7 将第一个待排序项插入数组的有序部分中

**15.13 算法 insertInOrder :** 终稿。这个算法不完全正确。仅当数组剩余部分有一个以上的项——即如果 `begin < end`——时, `else` 子句才起作用。比如说, 如果 `begin` 和 `end` 相等, 递归调用将等价于

```
insertInOrder(anEntry, a, begin, begin - 1);
```

而这是不正确的。

如果 `end` 和 `begin` 初始时不等, 那它们会相等吗? 会的。当 `anEntry` 小于 `a[begin], …, a[end]` 之间的所有项时, 每次递归调用都使 `end` 减 1, 直到最后 `end` 等于 `begin`。当发生这种情况时我们该怎么办? 因为有序部分中含有一个项 `a[end]`, 所以我们将 `a[end]` 移到下一个更高位置, 并将 `anEntry` 放到 `a[end]` 中。

这些修改反映在下列修改后的算法中。

```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// Revised draft.

if (anEntry >= a[end])
 a[end + 1] = anEntry
else if (begin < end)
{
 a[end + 1] = a[end]
 insertInOrder(anEntry, a, begin, end - 1)
}
else // begin == end and anEntry < a[end]
{
```

```

 a[end + 1] = a[end]
 a[end] = anEntry
}

```

## 插入排序的效率

回头看段 15.10 给出的迭代算法 `insertionSort`。对于有  $n$  项的数组，`first` 是 0 且 `last` 是  $n-1$ 。则 `for` 循环执行  $n-1$  次，那么方法 `insertInOrder` 被调用  $n-1$  次。所以，在 `insertInOrder` 中，`begin` 是 0 且 `end` 介于 0 到  $n-2$  之间。每次调用方法时，`insertInOrder` 内的循环最多执行 `end - begin + 1` 次。所以这个循环执行的总次数最多是

$$1+2+\cdots+(n-1)$$

次。这个和是  $n(n-1)/2$ ，故插入排序是  $O(n^2)$  的。执行递归插入排序时，与迭代插入排序有相同的操作，所以也是  $O(n^2)$  的。

这个分析提供的是最差情况。最优情况下，`insertInOrder` 内的循环将立即退出。如果数组已有序时会出现这样的情形。则最优情况下，插入排序是  $O(n)$  的。一般地，数组越有序，`insertInOrder` 内要做的事情越少。这个事实及它相对简单的实现过程，使得插入排序在数组不需要改变太多的应用中很受欢迎。例如，有些顾客数据库每天只增加很少比例的新顾客。

下一章当数组很小时使用插入排序。

 **注：插入排序的时间效率**  
插入排序最优时是  $O(n)$  的，最坏时是  $O(n^2)$  的。数组越接近有序，插入排序要做的工作越少。

## 结点链表上的插入排序

你常常会对数组进行排序，但有时可能也需要对结点链表进行排序。这种时候，插入排序是易于理解的一种方法。15.15

图 15-8 所示为结点中含有整数的一个链表，且已按升序有序。为了明白如何对链表进行插入排序，先假定我们想将一个结点插入这个链表中，并让结点中的整数仍保持有序。

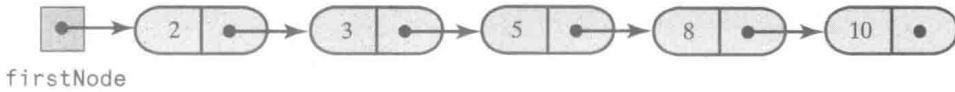


图 15-8 升序有序的整数链表

假定要插入链表中的结点含有整数 6。我们需要知道新结点在链表中的位置。因为引用 `firstNode` 指向链表中的首结点，所以我们可以从这里开始。向链表尾移动的过程中进行比较，直到找到正确的插入点为止。所以，我们将 6 与 2 进行比较，然后是与 3、与 5，最后是与 8 进行比较，发现 6 应该位于 5 和 8 之间。

要将一个结点插入链表中，需要一个指向插入点的前一结点的引用。所以对链表遍历的过程中，我们保存指向当前结点的前一个结点的引用，如图 15-9 所示。注意到，插入链表的开始位置时与插入链表的其他位置时有所不同。

现在假定，我们有方法 `insertInOrder(nodeToInsert)`，可将一个结点插入链表中正确的有序位置，如前所述。采用对数组进行排序的相同策略，然后可以用这个方法来实现插15.16

入排序：将链表划分为两部分。第一部分是有序的，初始时它仅含有第一个结点。第二部分是无序的，初始时它含有链表中其余的结点。图 15-10 说明了如何进行这个划分。先让变量 `unsortedPart` 指向第二个结点，然后将第一个结点的链接部分置为 `null`。

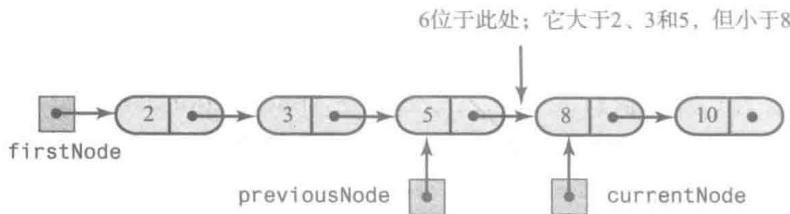


图 15-9 遍历链表时寻找插入点，保存指向当前结点的前一个结点的引用

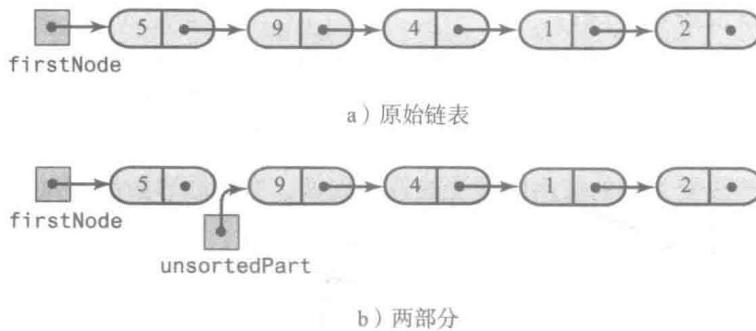


图 15-10 插入排序的第一步是将结点链表分为两部分

要排序结点，可以使用方法 `insertInOrder`，从无序部分取每个结点并将其插入有序部分中。注意到，我们是将已存在的结点重新链接上，而不是创建新结点。

15.17

为这个讨论再花点笔墨，假定我们要将一个排序方法添加到使用链表来表示某个集合的类 `LinkedGroup` 中。因为排序过程要求我们比较集合中的对象，所以排序方法必须属于实现了接口 `Comparable` 的类。所以类定义的开头如下所示。

```
public class LinkedGroup<T extends Comparable<? super T>>
{
 private Node firstNode;
 int length; // Number of objects in the group
 ...
}
```

回忆 Java 插曲 5，你可以在类定义的开头限定泛型数据类型。

这个类有一个内部类 `Node`，它对其私有数据域有设置方法和获取方法。下面的私有方法将 `nodeToInsert` 指向的结点插入 `firstNode` 指向的有序链表中。

```
private void insertInOrder(Node nodeToInsert)
{
 T item = nodeToInsert.getData();
 Node currentNode = firstNode;
 Node previousNode = null;
 // Locate insertion point
 while ((currentNode != null) &&
 (item.compareTo(currentNode.getData()) > 0))
 {
 previousNode = currentNode;
 currentNode = currentNode.getNextNode();
 } // end while
```

```

// Make the insertion
if (previousNode != null)
{ // Insert between previousNode and currentNode
 previousNode.setNextNode(nodeToInsert);
 nodeToInsert.setNextNode(currentNode);
}
else // Insert at beginning
{
 nodeToInsert.setNextNode(firstNode);
 firstNode = nodeToInsert;
} // end if
} // end insertInOrder

```

局部变量 `item` 的值是待插入结点的数据部分。`while` 循环中，将 `item` 与链表中每个结点的数据值进行比较，直到 `item` 小于等于一个数据值，或是到达链表尾时为止。然后使用引用 `previousNode` 和 `currentNode` 将给定结点插入合适的位置。

执行插入排序的方法如下所示。局部变量 `unsortedPart` 从第二个结点开始，之后在循环执行过程中指向链表其余的每个结点。这些结点中的每一个依次被插入链表的有序部分中。注意，`length` 是链表中结点的个数。15.18

```

private void insertionSort()
{
 // If fewer than two items are in the chain, there is nothing to do
 if (length > 1)
 {
 // Assertion: firstNode != null
 // Break chain into 2 pieces: sorted and unsorted
 Node unsortedPart = firstNode.getNextNode();
 // Assertion: unsortedPart != null
 firstNode.setNextNode(null);

 while (unsortedPart != null)
 {
 Node nodeToInsert = unsortedPart;
 unsortedPart = unsortedPart.getNextNode();
 insertInOrder(nodeToInsert);
 } // end while
 } // end if
} // end insertionSort

```

 **学习问题 3** 在前面的 `insertionSort` 方法中，如果你将语句行

```
unsortedPart = unsortedPart.getNextNode();
```

移到调用 `insertInOrder` 之后，方法还能奏效吗？请解释原因。

**学习问题 4** 前面的 `insertionSort` 方法不是静态方法。为什么？

在链表上进行插入排序的效率。对于含  $n$  个结点的链表，方法 `insertInOrder` 所进行的比较次数最多是链表中有序部分的结点个数。方法 `insertionSort` 调用 `insertInOrder` 共  $n-1$  次。第一次这样做时，有序部分含有一个项，所以进行了一次比较。第二次时，有序部分含有两个项，所以最多进行两次比较。继续这个过程，可以看到比较次数最多是

$$1+2+\dots+(n-1)$$

这个和是  $n(n-1)/2$ ，故插入排序是  $O(n^2)$  的。

 **注：**对结点链表进行排序可能是困难的。但插入排序提供了一个完成这个任务的合理方法。

## 希尔排序

15.20

到目前为止，我们讨论的排序算法都很简单且常用，但它们用于大数组时效率不高。希尔排序是插入排序的变体，能比  $O(n^2)$  更快。

在插入排序过程中，数组项只移动到相邻位置。当项与其正确的有序位置相距甚远时，它必须进行很多次这样的移动。所以当数组完全无序时，插入排序要花很多的时间。但当数组基本有序时，插入排序有很好的效率。事实上，段 15.14 已经说明了数组越有序，方法 `insertInOrder` 所做的工作越少。

利用这些观察结果，Donald Shell 在 1959 年设计了一个改进的插入排序，现在称为 **希尔排序**（Shell sort）。Shell 想让项移到比相邻项更远的位置。为此，他对具有等间距下标的项进行排序。不是将项移到相邻位置，而是移到几个位置之外。得到的结果是几乎有序的数组——可使用普通的插入排序进行高效率排序的数组。

15.21

例如，图 15-11 所示为一个数组及每隔 5 项组成的组。第一组含有整数 10、9 和 7；第二组含有 16 和 6；等等。刚好有 6 个这样的组。

现在使用插入排序分别对这 6 个项组中的每一个进行排序。图 15-12 所示为有序的组及由此得到的原数组的状态。注意到，数组比原始状态“更加有序”了。

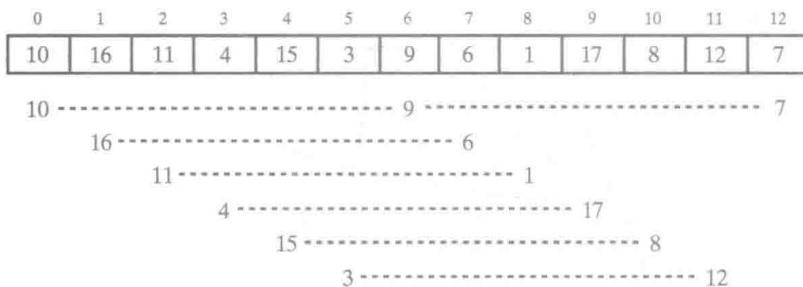


图 15-11 数组及下标间隔为 6 的项组成的组

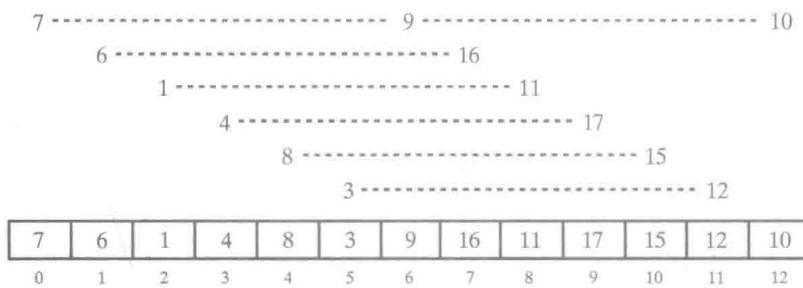


图 15-12 图 15-11 所示的每个组在排序后，及由它们组成的数组

15.22

现在我们形成了新的项组，但这次我们减小下标之间的间隔。Shell 建议，下标间的初始间隔是  $n/2$ ，且每趟排序中这个值减半直到为 1 时为止。我们的示例数组有 13 项，所以我们从间隔为 6 开始。现在将间隔减小到 3。图 15-13 所示为得到的项组，图 15-14 所示为排序后的项组。

将当前间隔 3 除以 2 得到 1。所以最后一步只是对整个数组进行普通的插入排序。这最后一步将对数组进行排序，不管之前我们对它做了什么。所以如果你使用任何的下标间隔，只要最后一个是 1，则希尔排序都能奏效。但不是任意序列都能使希尔排序有高效率，这一

点将在段 15.24 中讨论。

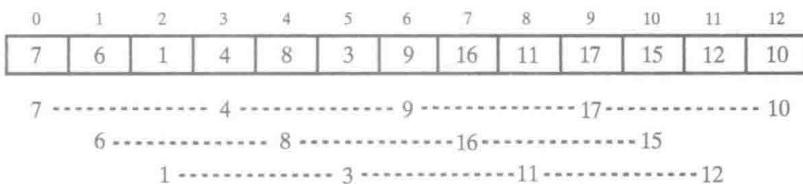


图 15-13 由图 15-12 所示数组中下标间隔为 3 的项组成的组

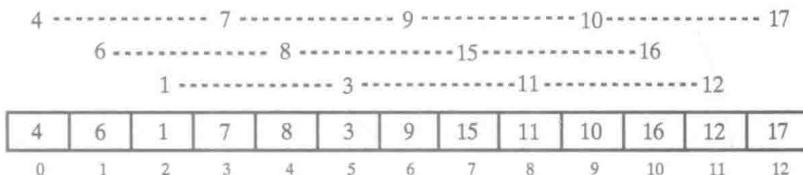


图 15-14 图 15-13 所示的每个组在排序后，及由它们组成的数组

 学习问题 5 使用希尔排序对含字母 I H B G E D F C A 的数组进行排序，下标间隔是 4、2 和 1。中间步骤是什么？

## 算法

希尔排序的核心是修改插入排序，以便其能对数组中有相等间距的项进行排序。将段 15.10 中给出的描述插入排序的两个算法组合起来并进行修改，我们得到下列对下标相距增量为 space 的数组项进行排序的算法。15.23

```
Algorithm incrementalInsertionSort(a, first, last, space)
// Sorts equally spaced array entries a[first] through a[last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length;
// space is the difference between the indices of the entries to sort.

for(unsorted = first + space 到 last 增量是 space)
{
 nextToInsert = a[unsorted]
 index = unsorted - space
 while ((index >= first) 且 (nextToInsert < a[index]))
 {
 a[index + space] = a[index]
 index = index - space
 }
 a[index + space] = nextToInsert
}
```

执行希尔排序的方法将调用 `incrementalInsertionSort`，并提供任一间隔因子序列。例如，下列算法使用段 15.22 描述的间隔。

```
Algorithm shellSort(a, first, last)
// Sorts the array entries a[first] through a[last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length.

n = 数组项的个数
space = n / 2
while (space > 0)
{
 for (begin = first 到 first + space - 1)
```

```

 {
 incrementalInsertionSort(a, begin, last, space)
 }
 space = space / 2
}

```



### 学习问题 6 使用希尔排序对下列数组进行升序排序，跟踪其步骤：9 6 2 4 8 7 5 3

## 希尔排序的效率

15.24

因为希尔排序重复地使用插入排序，当然好像是比只使用一次插入排序要做更多的工作。但实际上并不是这样的。虽然我们使用了若干次插入排序而不是仅用一次，但对数组最初的排序远比原来的数组要小得多，后来的排序是对部分有序的数组进行的，且最后的排序是对几乎全部有序的数组进行的。直观来看，这似乎是不错的。即使希尔排序不很复杂，但它的分析也十分复杂。

因为 `incrementalInsertionSort` 方法涉及一个循环，而本身又是在嵌套的循环内被调用，故希尔排序用到了 3 层嵌套的循环。这样的算法常常是  $O(n^3)$  的，但可以证明，希尔排序的最差情况仍是  $O(n^2)$  的。如果  $n$  是 2 的幂次，则平均情况是  $O(n^{1.5})$ 。如果稍稍调整一下间隔，能使希尔排序的效率更高。

一项改进是避免 `space` 是偶数值。图 15-11 提供了 `space` 为 6 的示例。例如，第一组含有 10、9 和 7。后来，将 `space` 分半，第一组含有 7、4、9、17 和 10，如图 15-13 所示。注意到，这两个组含有公共项，即 10、9 和 7。所以，当 `space` 是偶数时所进行的比较，会在当增量是 `space/2` 时的下一趟排序中重复。

为避免这种低效率，当 `space` 为偶数时，只需将其加 1。这个简单的修改能得到没有公因子的连续增量。希尔排序的最差情况则为  $O(n^{1.5})$ 。其他的 `space` 序列甚至能得到更高的效率，不过证明这一点仍是未解之谜。有理由选择改进的希尔排序对中等大小的数组进行排序。



### 注：希尔排序的时间效率

本章实现的希尔排序有  $O(n^2)$  的最差情况。当 `space` 为偶数时，将其加 1，则最差情况可以改进为  $O(n^{1.5})$ 。

## 算法比较

15.25

图 15-15 将本章提出的 3 种排序算法的时间效率进行了总结。一般地，选择排序是最慢的算法。利用插入排序最优情况的希尔排序是最快的。

|      | 最优情况     | 平均情况         | 最差情况         |
|------|----------|--------------|--------------|
| 选择排序 | $O(n^2)$ | $O(n^2)$     | $O(n^2)$     |
| 插入排序 | $O(n)$   | $O(n^2)$     | $O(n^2)$     |
| 希尔排序 | $O(n)$   | $O(n^{1.5})$ | $O(n^{1.5})$ |

图 15-15 以大  $O$  符号表示的 3 种排序算法的时间效率

## 本章小结

- 对数组的选择排序，选择最小的项并将其与第一项相交换。忽略新的第一项，排序找到数组中其余项中的最小项并将其与第二项相交换，以此类推。
- 一般地，迭代执行选择排序，虽然简单的递归形式也是可行的。
- 选择排序在所有情形下都是  $O(n^2)$  的。
- 插入排序将数组划分为两部分，有序的和未排序的。初始时，数组的第一项属于有序部分。排序找到下一个未排序的项，将它与有序部分中的项进行比较。持续进行比较时，每个有序项向数组尾的方向移动一个位置，直到找到未排序项的正确位置时为止。然后排序将项插入通过移动而腾出的正确位置。
- 可以用迭代或递归方式执行插入排序。
- 插入排序最差情况是  $O(n^2)$  的，但最好情形是  $O(n)$  的。数组越有序，插入排序要做的工作越少。
- 可以使用插入排序对结点链表进行排序，通常来讲对链表排序是困难的一件事。
- 希尔排序是插入排序的修改版，它对数组内具有相等间隔的项进行排序。这个机制能高效地重排数组，以使数组几乎有序，从而能使用普通插入排序快速完成工作。
- 本章所实现的希尔排序的最差情况是  $O(n^2)$  的。稍加修改，它的最差情况至少可改进为  $O(n^{1.5})$ 。

## 程序设计技巧

- 为能对任意类型使用 Comparable，用 Comparable<? super T> 替代 Comparable<T>。

## 练习

- 设有数组含有整数 5 7 4 9 8 5 6 3，当使用选择排序进行升序排序时，显示每趟的内容。
- 使用插入排序重做练习 1。
- 使用希尔排序重做练习 1。
- a. 写选择排序算法的伪代码，选择数组中有最大值而不是最小值的项，并将数组按降序排序。  
b. 使用你的算法重做练习 1。  
c. 修改段 15.5 给出的迭代方法 selectionSort，以实现你的算法。
- 重做练习 4，这次将数组按升序排序。
- 修改段 15.5 给出的迭代方法 selectionSort，让其参数为 first 和 last，而不是 n。
- 修改选择排序算法，每趟它找到数组未排序部分的最大值和最小值。然后排序将它们与数组项进行交换将其放到正确位置。
  - 排序 n 个值时共需进行多少次比较？
  - a 中的答案大于、小于或等于普通选择排序所需的比较次数吗？
- 冒泡排序（bubble sort）可将含 n 项的数组通过  $n-1$  趟扫描进行升序排列。每一趟中，它比较相邻项，如果它们呈逆序则交换它们。例如，第一趟中，它比较第一项和第二项，然后比较第二项和第三项，以此类推。在第一趟的最后，最大项位于数组的最后，也处于它的合适位置。我们说，它已经冒泡到正确的位置了。后续的每一趟忽略数组最后的项，因为它们已经有序且大于所剩的项。所以每趟都比前一趟进行更少的比较。图 15-16 给出冒泡排序的示例。
  - 采用迭代方式实现冒泡排序；
  - 采用递归方式实现冒泡排序。

原始数组

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 8 | 2 | 6 | 4 | 9 | 7 | 1 |
|---|---|---|---|---|---|---|

1 趟扫描后

有序部分

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 8 | 7 | 1 | 9 |
|---|---|---|---|---|---|---|

2 趟扫描后

有序部分

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 7 | 1 | 8 | 9 |
|---|---|---|---|---|---|---|

3 趟扫描后

有序部分

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 1 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|

4 趟扫描后

有序部分

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 4 | 1 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|

5 趟扫描后

有序部分

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|

6 趟扫描后

有序部分

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|

图 15-16 数组的冒泡排序（见练习 8）

9. 练习 8 中的冒泡排序与本章其他排序算法相比，效率如何？
10. 练习 8 中的冒泡排序永远进行  $n$  趟。但是，在完成所有  $n$  趟前，数组可能已经有序。例如，对如下数组进行冒泡排序
- 9 2 1 6 4 7 8  
两趟之后数组已有序：  
2 1 6 4 7 8 9 (1 趟结束后)  
1 2 4 6 7 8 9 (2 趟结束后)
- 但是，因为在第二趟中发生了交换，故排序还需要再进行一趟扫描以检查数组是否有序。其他的扫描，例如练习 8 中的算法要做的，就不是必要的了。
- 通过记下最后交换的位置，可以跳过这些不必要的扫描，从而少做工作。第一趟中，最后交换的是 9 与 8。第二趟将检查到 8。但第二趟扫描中，最后交换的是 6 和 4。现在知道 6、7、8 和 9 都已有序。第三趟只需检查到 4，而不是原始的冒泡排序所执行的检查到 7。第三趟扫描中没有发生交换，所以这趟扫描中最后交换的下标是 0，表示不再需要扫描。实现这个修改的冒泡排序。
11. 设计一个算法，检查所给的数组是否升序有序。写一个实现你算法的 Java 方法。可以用这个方法来测试排序方法是否正确执行。
12. 假定想对 `Comparable` 对象的集合执行选择排序。集合是类 `Group` 的实例。
- 你需要哪个私有方法？
  - 实现对 `Group` 实例的对象进行选择排序的方法。
  - 使用大  $O$  符号，描述方法的时间效率。
13. 本章中的哪个递归算法是尾递归？
14. 如段 15.24 所述，当 `space` 是偶数时，可以通过将其加 1，来改进希尔排序的效率。
- 找几个示例进行验证，说明连续的增量没有公因子。
  - 当 `space` 为偶数时，让其减 1，不会得到没有公因子的连续增量。找到含  $n$  项的例子，说明这

个现象。

c. 修改段 15.23 给出的希尔排序算法，让 `space` 不是偶数。

15. 假定你要找到含  $n$  项无序数组中的最大项。算法 A 顺序查找整个数组，记录下到目前为止看到的最大项。算法 B 将数组降序排序，然后报告第一项为最大项。比较这两个方法的时间效率。

16. 考虑段 15.10 给出的算法 `insertInOrder`，将一个对象插入数组有序部分的正确位置。如果使用一个类似的算法，将结点插入有序结点链表中，我们应该从链表表尾开始。例如，为将含有 6 的结点插入图 15-8 所示的链表中，首先应该比较 6 与 10。因为 6 应被放在 10 的前面，所以应该比较 6 与 8。因为 6 应该被放在 8 的前面，所以比较 6 与 5，发现 6 应该被放在 5 和 8 之间。

描述如何用这个算法来定义对结点链表进行排序的方法 `insertInOrder`。你的方法是时间上有效的吗？

17. 考虑类 `Person`，它含有字符串型的私有数据域 `phoneNumber`。电话号码中有一个用破折号表示的可选的地区码。例如，两个号码 443-555-1232 和 555-0009 都是可能的电话号码。为类 `Person` 写方法 `compareTo`，能让 `Person` 对象的数组按电话号码排序。

18. 考虑类 `Student`，它含有用于名字、班级排名、学号和平均成绩的私有数据域。假定，对于 `Student` 对象的数组，你想根据前面所列的任一个数据域进行排序。

a. 实现这样的排序时遇到的困难是什么？

b. 这个问题的一种解决方案是，为每种排序标准定义一个新类。每一个这种类都封装一个 `Student` 对象。这种方式下，可以使用本章所给的排序方法。为实现这个方案的其他程序员提供必要的设计细节。

c. 这个问题的另一种解决方案是，修改排序方法的签名和定义。方法的一个参数是一个对象，这个方法可以根据某个标准比较两个 `Student` 对象。这个参数可以是对应于排序标准的任一个新类的对象。为实现这个方案的其他程序员提供必要的设计细节。

## 项目

1. 当你想说明算法的行为时，各种排序算法的演示都是有益的。考虑一组不同长度的竖直线，如图 15-17a 所示。创建一个排序演示，将线按长度排序，如图 15-17b 所示。应该画出排序算法进行每次交换或移动后线的状态。如果每次重画后稍稍延迟运行，则会得到排序的动画。

可以从画 256 根线开始，每根线有 1 个像素宽，但有不同的长度——可能还有不同的颜色——将其从最短到最长排列，使其看起来像是一个三角。然后用户可以弄乱这些线。用户给出命令后，排序算法应该排序这些线。

可以为每个排序算法单独提供一个演示，或许是小程序。或者，可以将所有算法含在一个程序中，告诉用户来选择一个算法。每个排序应该从同一组弄乱的线开始，这样用户可以比较这些方法。也可以随机选择一个排序算法，看看用户能不能猜中是哪个算法。

2. 实现插入排序和希尔排序，并统计排序过程中进行的比较次数。使用你的实现，对不同大小的随机产生的 `Integer` 对象数组进行排序，比较这两种排序方法。另外，将段 15.23 实现的希尔排序，与当 `space` 是偶数时将其加 1 而进行修改的希尔排序进行比较。多大的数组会使比较次数明显不同？这个大小与算法的大  $O$  表示所预测的一致吗？

3. 实现本章所给的排序算法。使用你的实现，比较不同的含 50 000 个随机 `Integer` 对象的数组的运

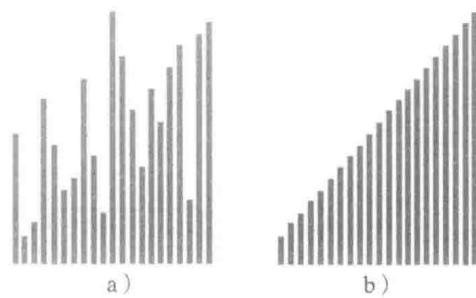


图 15-17 对不同的线进行排序的动画演示的初始和最终图像

行时间。见第 4 章结尾处描述的如何对一段 Java 代码进行计时的项目。写个总结，哪个算法更高效？为什么？

4. 考虑一个  $n \times n$  的整数数组。
  - a. 写出将数组行按它的第一个值进行排序的算法。
  - b. 使用大  $O$  符号，描述上述算法的效率。
  - c. 实现你的算法。
5. 假定你要在链表上执行希尔排序。
  - a. 修改算法 `incrementalInsertionSort`，以便它能用于链表而不是数组。
  - b. 比较 `incrementalInsertionSort` 用于数组及用于链表的性能。
  - c. 使用修改后的算法，实现链表上的希尔排序。
  - d. 找出对  $n$  个不同值的链表进行排序所需的时间。（见第 4 章结尾处描述的如何对一段 Java 代码进行计时的项目。）画出运行时间与  $n$  的关系图。
  - e. 假定你的排序的性能是  $O(n^k)$  的，评估  $k$  的值。
6. **计数排序 (counting sort)** 是对含  $n$  个从 0 到  $m$  (含) 之间的正整数的数组进行排序的一种简单方法。你需要  $m+1$  个计数器。然后，只需扫描数组一趟，计下数组中每个整数出现的次数。例如，图 15-18 显示了一个含 0 ~ 4 之间整数的数组，及计数排序对数组进行一趟扫描后的 5 个计数器。从计数器中可以看出，数组中含有一个 0、3 个 1、2 个 2、1 个 3 及 3 个 4。这些结果能确定有序数组应该含有 0 1 1 1 2 2 3 4 4 4。
  - a. 写出执行计数排序的方法。
  - b. 使用大  $O$  符号，描述这个算法的效率。
  - c. 与插入排序相比，计数排序的效率如何？
  - d. 这个算法能用于一般的排序算法吗？解释之。

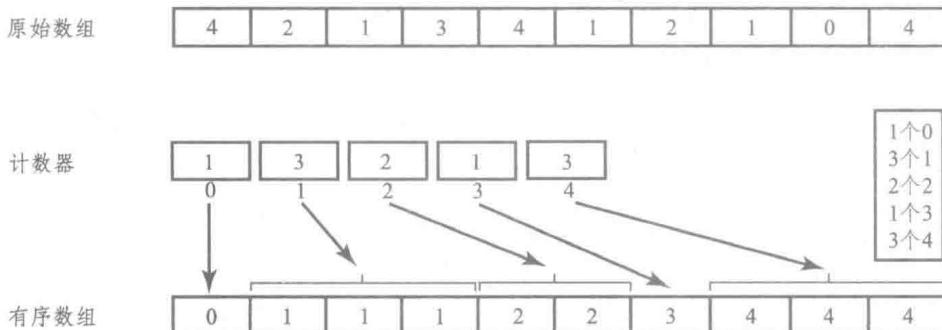


图 15-18 数组的计数排序（见项目 6）

7. a. 重复第 9 章项目 2，使用迭代的基于栈的方式替代递归方式实现算法。
- b. 考虑算法的下列版本。将  $S_1$  的栈顶移到  $S_2$  后，比较  $S_1$  的新栈顶项  $t$  与  $S_2$  的栈顶项和  $S_3$  的栈顶项。然后，或者将项从  $S_2$  移到  $S_3$ ，或从  $S_3$  移到  $S_2$ ，直到为  $t$  找到正确位置。将  $t$  入  $S_2$  中。继续这个过程，直到  $S_1$  为空时为止。最后，将留在  $S_3$  中的所有项移回  $S_2$ 。用迭代方式使用基于栈的方法实现这个修改算法。

# 更快的排序方法

先修章节：第 4 章、第 9 章、Java 插曲 5、第 15 章

## 目标

学习完本章后，应该能够

- 使用下列方法将一个数组按升序排序：归并排序、快速排序和基数排序
- 评估排序的效率，讨论不同方法的相对效率

当你想排序小数组时，第 15 章看到的排序算法通常就足够用了。甚至如果你想对更大的数组排序一次，那些算法可能也是一个合理的选择。另外，插入排序是对结点链表进行排序的好方法。但是，当你需要频繁地对非常大的数组进行排序时，那些方法要花太多的时间。本章介绍几个排序算法，通常来讲它们比第 15 章的方法要快得多。这里给出的每个算法，将数组按升序排序。不过，简单修改一下，每一个算法都能用于数组的降序排序。

## 归并排序

归并排序（merge sort）将数组分为两半，分别对两半进行排序，然后将它们合并为一个有序数组。<sup>16.1</sup> 归并排序的算法常常用递归方式描述。回忆一下，递归算法用同一问题的更小版本来表示解决问题的方案。当你将问题划分为两个或多个更小但不同的问题时，解决每一个新问题，然后将它们的方案合并为解决原始问题的方案，这个策略称为分治法（divide and conquer）算法。即将问题划分为小块，然后攻克每个小块以达成解决方案。虽然分治法算法常常用递归表示，但也不是必须要用递归。

当采用递归表示时，分治法算法含有两个或多个递归调用。到目前为止你见过的大多数递归方案没有使用分治策略。例如，前一章段 15.6 给出了选择排序的一个递归版本。尽管算法考虑越来越小的数组，但也没有将问题划分为两个排序问题。

执行归并排序时要做的实际工作都在归并步骤，这也是程序设计时的主要工作，所以我们从这里开始。

## 归并数组

假定你有两个不同的有序数组。归并两个有序数组并不困难，但它需要一个另外的数组。两个数组都是从开头处理到末尾，将一个数组中的项与另一个数组中的项进行比较，将较小的项拷贝到新的第三个数组中，如图 16-1 所示。当到达一个数组的末尾之后，只需将另一个数组中的剩余项拷贝到新的第三个数组中即可。<sup>16.2</sup>

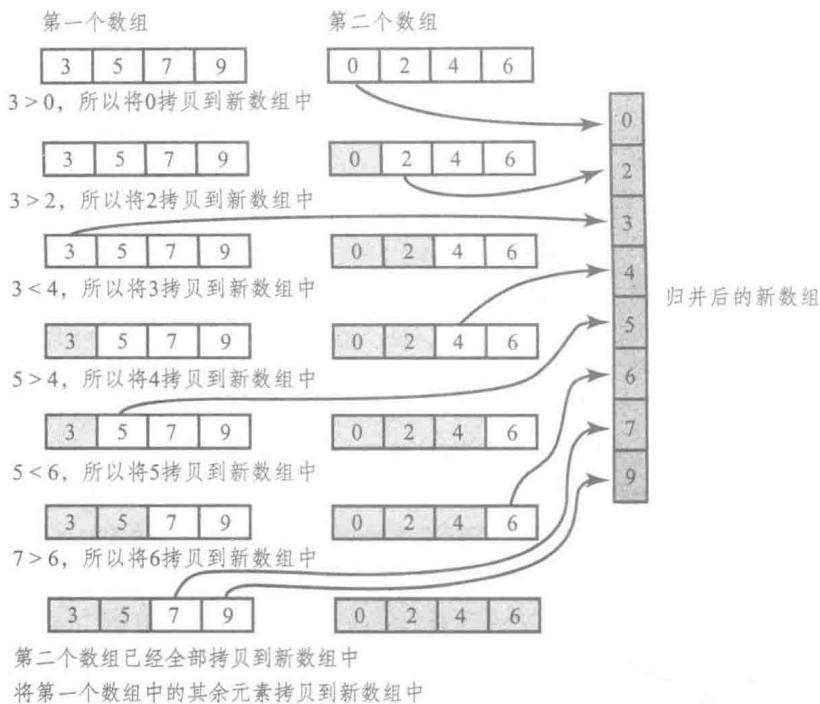


图 16-1 将两个有序数组归并为一个有序数组

### 递归的归并排序

**16.3 算法。**在归并排序中，归并了两个有序数组，实际上它们是原始数组的两半。即将数组一分为二，排序每一半，然后将这两段有序部分合并到第二个临时数组中，如图 16-2 所示。然后将临时数组拷贝回原数组中。



图 16-2 归并排序中的主要步骤

这个设计听上去挺简单的，但如何排序数组的两半呢？当然是使用归并排序！如果 `mid` 是含  $n$  项数组中间项的下标，则我们必须对下标从 0 到 `mid` 间的项进行排序，然后对下标从 `mid+1` 到  $n-1$  间的项进行排序。因为这些排序又是对归并排序算法的递归调用，所以算法需要两个参数——`first` 和 `last`——来标出待排序数组范围的第一个和最后一个下标。使用记号  $a[first..last]$  表示数组项  $a[first], a[first + 1], \dots, a[last]$ 。

归并排序有下列伪代码：

```
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first..last] recursively.
```

```

if (first < last)
{
 mid = first 和 last 的中间点
 mergeSort(a, tempArray, first, mid)
 mergeSort(a, tempArray, mid + 1, last)
 使用数组 tempArray 合并有序的两半 a[first..mid] 和 a[mid + 1..last]
}

```

注意到，算法没有处理少于等于一个项的数组。

下列伪代码描述了归并步骤。

```

Algorithm merge(a, tempArray, first, mid, last)
// Merges the adjacent subarrays a[first..mid] and a[mid + 1..last].
beginHalf1 = first
endHalf1 = mid
beginHalf2 = mid + 1
endHalf2 = last
// While both subarrays are not empty, compare an entry in one subarray with
// an entry in the other, then copy the smaller item into the temporary array
index = 0 // Next available location in tempArray
while ((beginHalf1 <= endHalf1) 且 (beginHalf2 <= endHalf2))
{
 if (a[beginHalf1] <= a[beginHalf2])
 {
 tempArray[index] = a[beginHalf1]
 beginHalf1++
 }
 else
 {
 tempArray[index] = a[beginHalf2]
 beginHalf2++
 }
 index++
}
// Assertion: One subarray has been completely copied to tempArray.

```

将另一个子数组中的剩余项拷贝到 tempArray 中

将 tempArray 中的项拷贝到数组 a 中

**跟踪算法中的步骤。**让我们来看看当对数组的两半调用 `mergeSort` 时会发生什么。图 16-3 显示了 `mergeSort` 将数组分为两半，然后递归地将每一半再分为两半，直到每一半只含一个项时为止。算法到此时，开始合并步骤。一对儿含一个项的子段合并为含两个项的子段。一对儿含两个项的子段合并为含 4 个项的子段，以此类推。图中箭头上的数字表示递归调用及进行合并的次序。

16.4

注意到，第一次合并发生在 4 次递归调用 `mergeSort` 之后及其他递归调用 `mergeSort` 之前。所以，递归调用 `mergeSort` 与调用 `merge` 是交织在一起的。真正的排序是发生在合并步骤而不是发生在递归调用步骤。正如你将看到的，这些结论可以用在两个方面。首先，我们能确定算法的效率。其次，可以迭代描述 `mergeSort` 算法。



注：归并排序在合并步骤中重排数组中的项。



学习问题 1 跟踪归并排序对数组 9 6 2 4 8 7 5 3 进行升序排序的步骤。

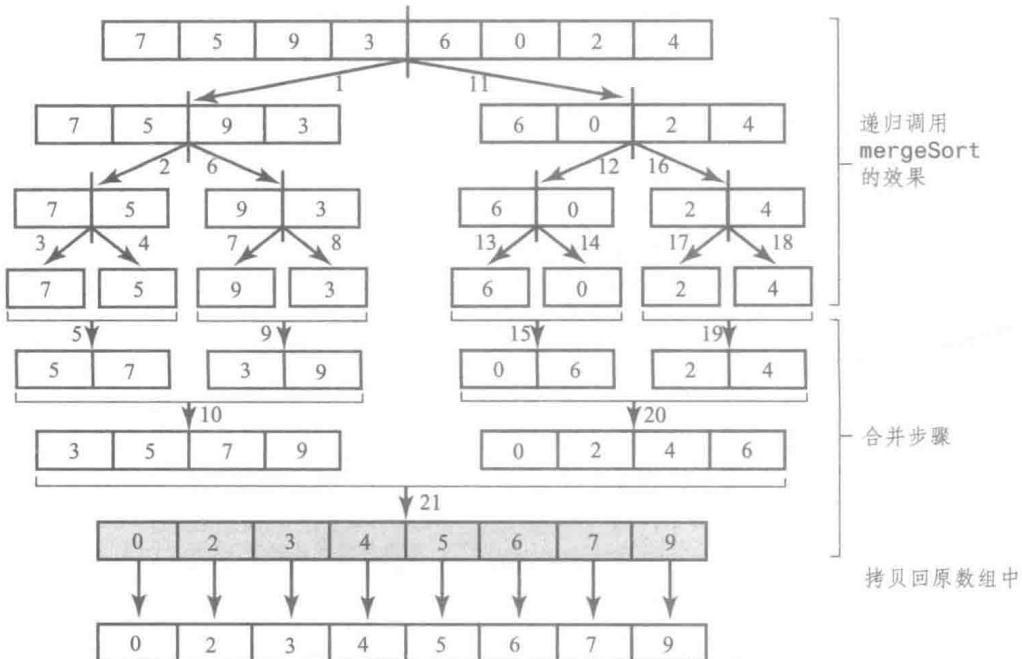


图 16-3 归并排序过程中递归调用及合并的效果

16.5

**实现说明。**虽然 `mergeSort` 的递归实现很简单，但注意应该只分配一次临时数组。因为数组是一个实现细节，所以你或许会冒险将空间分配隐藏在方法 `merge` 中。但是，因为每次递归调用 `mergeSort` 时都会调用 `merge`，故这个方式会导致临时数组被分配及被初始化很多次。相反，我们可以在下列公有的 `mergeSort` 方法中分配一个临时数组，然后将它传给私有的 `mergeSort` 方法，前面已经给出了这个方法的伪代码：

```
public static <T extends Comparable<? super T>>
 void mergeSort(T[] a, int first, int last)
{
 // The cast is safe because the new array contains null entries
 @SuppressWarnings("unchecked")
 T[] tempArray = (T[])new Comparable<?>[a.length]; // Unchecked cast
 mergeSort(a, tempArray, first, last);
} // end mergeSort
```

Java 插曲 5 介绍了记号 `? super T` 表示的是 `T` 的任意超类。当我们分配 `Comparable` 对象的数组时，使用通配符 `? super T` 来表示任意的对象。然后将数组转型为类型 `T` 对象的数组。

本章最后的项目 1 要求你实现递归的归并排序。

## 归并排序的效率

16.6

假定现在  $n$  是 2 的幂次，这样我们就可以一直用 2 除以  $n$ 。图 16-3 中的数组有  $n=8$  个项。第一次调用 `mergeSort`，又导致两次递归调用 `mergeSort`，将数组分为两个各含  $n/2$ （即 4）个项的子段。两次递归调用 `mergeSort` 中的每一次，又导致两次递归调用 `mergeSort`，将两个子段划分为 4 个各含  $n/2^2$ （即 2）个项的子段。最后，递归调用 `mergeSort` 将 4 个子段分为 8 个各含  $n/2^3$ （即 1）个项的子段。进行三层递归调用，获得各含一个项的子段。注意到，原来的数组含有  $2^3$  个项。指数 3 是递归调用的层数。一般地，如果  $n$  是  $2^k$ ，就会发生  $k$  层递归调用。

现在考虑合并步骤，因为这是真正工作之所在。在共有  $n$  个项的两个子段中，合并步骤最多进行  $n-1$  次比较。图 16-4 显示的是需要  $n-1$  次比较的合并示例，而图 16-1 显示的是少于  $n-1$  次比较的示例。每次合并还需要移向临时数组的  $n$  次移动及移回原数组的  $n$  次移动。总共，每次合并最多需要  $3n-1$  次操作。

每次对 `mergeSort` 的调用还要调用 `merge` 一次。最初对 `mergeSort` 的那次调用，合并操作最多需要  $3n-1$  次操作。这是  $O(n)$  的。这个合并的例子如图 16-3 中第 21 步所示。两次递归调用 `mergeSort` 导致两次调用 `merge`。每次调用最多用  $3n/2-1$  次操作合并  $n/2$  项。然后两次合并最多需要  $3n/2-2$  次操作。它们是  $O(n)$  的。下一层递归调用  $2^2$  次 `mergeSort`，导致 4 次调用 `merge`。每次调用 `merge` 时最多用  $3n/2^2-1$  次操作合并  $n/2^2$  个项。这 4 次合并一起，最多使用  $3n/2^2$  次操作，所以也是  $O(n)$  的。

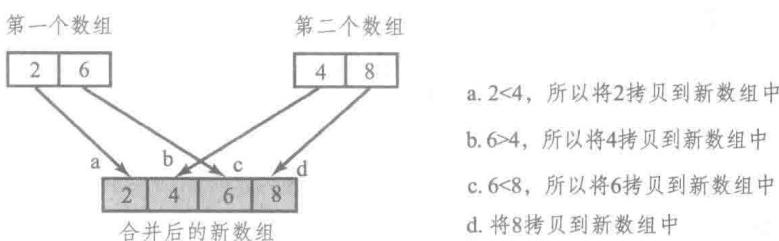


图 16-4 合并两个有序数组的最差情况

如果  $n$  是  $2^k$ ，递归调用 `mergeSort` 方法  $k$  层，导致进行  $k$  层合并。每一层的合并都是  $O(n)$  的。因为  $k$  是  $\log_2 n$  的，所以 `mergeSort` 是  $O(n \log n)$  的。当  $n$  不是 2 的幂次时，可以找到整数  $k$ ，满足  $2^{k-1} < n < 2^k$ 。例如，当  $n$  是 15 时， $k$  是 4。所以

$$k-1 < \log_2 n < k$$

如果对  $\log_2 n$  向上取整，可得到  $k$ 。故这种情形下归并排序也是  $O(n \log n)$  的。注意到，不管数组的初始状态如何，合并步骤都是  $O(n)$  的。则最差、最优及平均情况下，归并排序都是  $O(n \log n)$  的。

归并排序的缺点是在合并阶段需要一个临时数组。在第 15 章的开头部分，我们提到过按高度对书架上的书进行排序。我们不需要另一个书架或是地板来充当额外的空间就能这样做。但现在你看到了，归并排序则需要这个额外的空间。本章后面，将看到另一个排序算法，它也花费  $O(n \log n)$  的时间，但不需要第二个数组。

### 注：归并排序的时间效率

归并排序在所有情形下都是  $O(n \log n)$  的。它对临时数组的需求是它的缺点。

另一种评估效率的方法。在第 9 章，我们使用递推关系来估算递归算法的时间效率。这里也可以使用这项技术。如果  $t(n)$  表示 `mergeSort` 的最差情况时间需求，则两次递归调用中各需要时间  $t(n/2)$ 。合并步骤是  $O(n)$  的。所以我们有下列推导：

$$\begin{aligned} t(n) &= t(n/2) + t(n/2) + n \\ &= 2 \times t(n/2) + n \quad \text{当 } n > 1 \text{ 时} \\ t(1) &= 0 \end{aligned}$$

解这个递推关系的第一步，是对某个  $n$  值估算它。因为  $t(n)$  涉及  $n/2$ ，所以选择  $n$  为 2 的幂次——例如 8——比较方便。则有

$$\begin{aligned}t(8) &= 2 \times t(4) + 8 \\t(4) &= 2 \times t(2) + 4 \\t(2) &= 2 \times t(1) + 2 = 2\end{aligned}$$

通过重复进行替换，对  $t(8)$  得到下列求解过程：

$$\begin{aligned}t(8) &= 2 \times t(4) + 8 \\&= 2 \times [2 \times t(2) + 4] + 8 \\&= 4 \times t(2) + 8 + 8 \\&= 4 \times 2 + 8 + 8 \\&= 8 + 8 + 8 \\&= 8 \times 3\end{aligned}$$

因为  $8=2^3$ ,  $3=\log_2 8$ , 所以我们猜测

$$t(n) = n \log_2 n$$

正如第 9 章所做的一样，现在我们需要证明，我们的猜测事实上是正确的。将这个证明留作练习。

## 迭代的归并排序

**16.8** 一旦有合并算法，开发递归的归并排序就容易了。开发迭代的归并排序却并不简单。我们从讨论递归方案入手。

递归调用只是将数组划分为  $n$  个只含一个项的子数组，如图 16-3 所示。虽然我们不需要递归方法将数组中的各项隔离开，但递归可以控制合并过程。要使用迭代代替递归，我们需要控制合并过程。这样一个算法不管是时间还是空间，都比递归算法更高效，因为它消除了递归调用，因此去掉了活动记录的栈。但迭代的归并排序更难写出没有错误的代码。

基本上，迭代的归并排序从数组头开始，将一对对的含单个项的字段合并为含两个项的字段。然后再返回到数组头，将一对对的含两个项的字段合并为含 4 个项的字段，以此类推。但是，合并某个长度的所有字段对后，可能还剩余若干项。合并这些项时需要格外小心。本章末尾的项目 2 要求你开发迭代的归并排序。那时你会看到，合并过程中，可以节省很多将临时数组复制回原数组所需的时间。

## Java 类库中的归并排序

**16.9** `java.util` 包中的类 `Arrays` 定义了静态方法 `sort` 的几个不同版本，用来将数组按升序排序。对于对象数组，`sort` 使用归并排序。方法

```
public static void sort(Object[] a)
```

将对象数组 `a` 的全部内容进行排序，而方法

```
public static void sort(Object[] a, int first, int after)
```

对 `a[first]` 到 `a[after-1]` 之间的对象进行排序。对这两个方法，数组中的对象必须定义了 `Comparable` 接口。

如果数组左半段中的项都不大于右半段中的项，则这些方法中使用的归并排序会跳过合并步骤。因为两段都已经有序，所以这种情形下合并步骤是不需要的。



**学习问题 2** 修改段 16.3 给出的归并排序算法，以便跳过任何不必要的合并过程，如刚才所述。



### 注：稳定的排序

如果排序算法不改变相等对象的相对次序，则称为稳定的（stable）。例如，如果数据集中对象  $x$  位于对象  $y$  之前，且  $x.\text{compareTo}(y)$  是 0，则稳定的排序算法在对数据进行排序后，对象  $x$  仍保持位于对象  $y$  之前。对某些应用来说，稳定性很重要。例如，假定你对一组人先按名字然后按年龄进行排序。则稳定的排序算法能够保证相同年龄的人将按字母序排列。

Java 类库中的归并排序是稳定的。本章末尾的练习 9 要求你指出本章和第 15 章给出的稳定的排序算法。

## 快速排序

现在来看另一个使用分治法策略的数组排序。快速排序（quick sort）将数组划分为两部分，但与归并排序不同，这两部分不一定是数组的一半。而是，快速排序选择数组中的一个项——称为枢轴（pivot）——来重排数组项，满足

- 枢轴所处的位置就是在有序数组中的最终位置
- 枢轴前的项都小于等于枢轴
- 枢轴后的项都大于等于枢轴

这个排列称为数组的划分（partition）。

创建划分将数组分为两部分，我们称为较小部分和较大部分，它们由枢轴分开，如图 16-5 所示。因为较小部分中的项小于等于枢轴，而较大部分中的项大于等于枢轴，故枢轴位于有序数组中正确且最终的位置上。现在如果我们对较小部分和较大部分两个子段进行排序——当然使用快速排序——则原数组将是有序的。下列算法描述了排序策略：

```
Algorithm quickSort(a, first, last)
// Sorts the array entries a[first..last] recursively.
if (first < last)
{
 选择枢轴
 基于枢轴划分数组
 pivotIndex = 枢轴的下标
 quickSort(a, first, pivotIndex - 1) // Sort Smaller
 quickSort(a, pivotIndex + 1, last) // Sort Larger
}
```

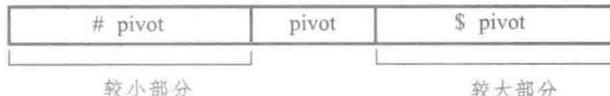


图 16-5 快速排序中数组的划分

## 快速排序的效率

注意到，创建划分——它完成了 quickSort 的大部分工作——在递归调用 quickSort 之前进行。这一点与归并排序不同，它的大部分工作是在递归调用 mergeSort 之后的合并

16.10

16.11

步骤完成的。划分过程需要不超过  $n$  次的比较，故与合并一样，它将是  $O(n)$  的任务。所以可以评估快速排序的效率，虽然我们尚未开发划分策略。

当枢轴移动到数组的中心时是理想情形，这样划分形成的两个子数组有相同的大小。如果对 `quickSort` 的每次递归调用都划分了大小相等的子数组，则快速排序与归并排序一样递归调用数组的两半。所以快速排序将是  $O(n \log n)$  的，这是最优情况。

但是这种理想情形或许不会总发生。最差情况下，每次划分都有一个空子数组。虽然一个递归调用什么也不做，但另一个调用必须排序  $n-1$  个项而不是  $n/2$  个项。结果是有  $n$  层递归调用而不是  $\log n$  层。所以最差情况下，快速排序是  $O(n^2)$  的。

所以枢轴的选择影响快速排序的效率。如果数组已经有序或接近有序，有些选择枢轴的机制可能导致最差情况。实际中，出现接近有序数组的情形，可能会比你想象的更频繁。后面你会看到，我们选择枢轴的机制对有序数组可以避免最差情况。

虽然我们不去证明它，但快速排序在平均情况下是  $O(n \log n)$  的。归并排序总是  $O(n \log n)$  的，而实际中，快速排序可能比归并排序更快，且不需要归并排序中合并操作所需的额外内存。



### 注：快速排序的时间效率

快速排序在平均情况下是  $O(n \log n)$  的，但在最差情况下是  $O(n^2)$  的。枢轴的选择将影响它的行为。

## 创建划分

16.12

选择图 16-5 中的枢轴并创建划分可以有不同的策略。假定现在你已经选定了一个枢轴，我们来看看如何不依赖于选择枢轴的策略而创建划分。后面实际采用的选择枢轴的机制，对这个划分过程会有小小的修改。

在选择一个枢轴后，将它与数组的最后项相交换，这样在创建划分时枢轴不会碍你的事。图 16-6a 显示了这一步骤之后的数组。从数组头开始，向数组尾方向移动（图中从左到右），查看第一个大于等于枢轴的项。在图 16-6b 中，这个项是 5，出现在下标是 `indexFromLeft` 的位置。用类似的方式，从倒数第二项开始，向数组头方向移动（图中从右向左），查看第一个小于等于枢轴的项。在图 16-6b 中，这个项是 2，出现在下标是 `indexFromRight` 的位置。现在，如果 `indexFromLeft` 小于 `indexFromRight`，则交换这两个下标处的项。图 16-6c 显示这一步骤后的结果。小于枢轴的 2，已经移向数组的开头，而大于枢轴的 5 已经移向相反的方向。

继续从左和从右开始的查找。图 16-6d 显示从左开始的查找停止在 4，而从右开始的查找停止在 1。因为 `indexFromLeft` 小于 `indexFromRight`，所以交换 4 和 1。现在数组如图 16-6e 所示。等于枢轴的项可以放在划分的任一边。

再继续查找。图 16-6f 显示从左开始的查找停止在 6，而从右开始的查找将越过 6 停止在 1。因为 `indexFromLeft` 不小于 `indexFromRight`，故不需要交换，且查找结束。剩下的唯一步骤是交换 `a[indexFromLeft]` 和 `a[last]`，将枢轴放在较小部分子数组和较大部分子数组的中间，如图 16-6g 所示。完成后的划分如图 16-6h 所示。

注意，前面的查找不能越过数组尾。稍后在段 16.15，你会看到实现这种需求的一种方便的方法。

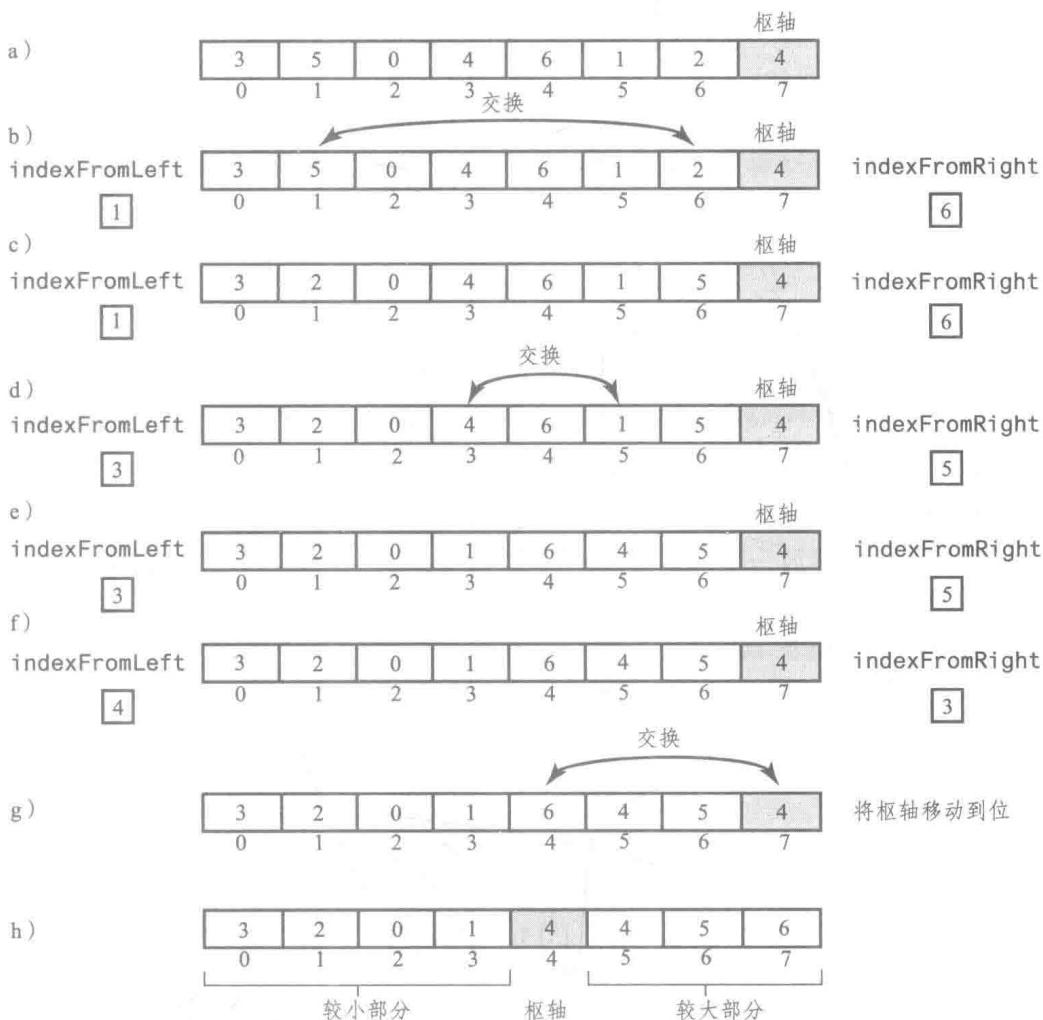


图 16-6 快速排序的划分机制

等于枢轴的项。注意到，较小部分和较大部分子数组中，都可能含有等于枢轴的项。你或许对此有些奇怪。为什么不总是将等于枢轴的项放到同一个子数组中呢？这样的策略能让一个子数组大于另一个。但是，为了提升快速排序的性能，我们想让子数组尽可能地等长。

注意到，从左开始的查找和从右开始的查找，在它们遇到等于枢轴的项时都会停止。这意味着，这样的项不是放在原地，而是要进行交换。这也意味着，这样的项有机会放在任一个子数组中。

**枢轴的选择。**理想地，枢轴应该是数组的中位值，所以较小部分和较大部分子数组都有相等——或接近相等——的项数。找到中位值的一种方法是排序数组，然后选择位于中间的值。但数组排序是原始问题，所以这个循环逻辑注定是失败的。找到中位值的其他方法太慢不能用。

因为选择最好的枢轴要花太多的时间，故我们至少应该试着避开坏的枢轴。所以不是找到数组中所有值的中位值，而是找到数组中这3个项的中位值：第一项、中间项及最后一项。完成这个任务的一个办法是，仅将这3个项进行排序，使用这3个项的中间值作为枢轴。图16-7显示的是，数组在其第一项、中间项及最后一项排序前后的情形。枢轴是5。这个枢轴选择策略称为三元中值枢轴选择 (median-of-three pivot selection)。

16.13

16.14

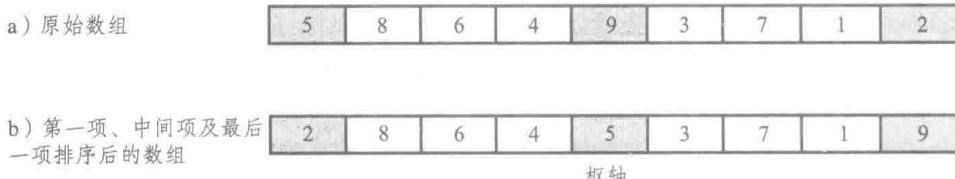


图 16-7 三元中值枢轴选择

 **注：**三元中值枢轴选择避免了快速排序当给定数组已经有序或接近有序时的最差情况性能。但理论上，不能避免其他情况下数组的最差性能，这样的性能在实际中不太可能出现。

16.15

**修改划分算法。**三元中值枢轴选择策略暗示，对划分机制要做小的修改。之前，我们在划分前将枢轴与数组的最后一项相交换。但这里，数组的第一项、中间项及最后一项已有序，所以我们知道，最后一项至少与枢轴一样大。所以，最后一项应该放在较大部分子数组中。我们只简单地让最后一项留在原地。为使枢轴不碍事，我们可以将它与倒数第二项  $a[\text{last}-1]$  相交换，如图 16-8 所示。所以，划分算法中从右开始的查找从下标  $\text{last}-2$  处开始。

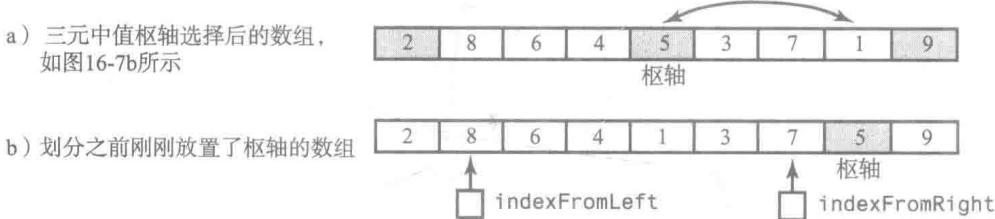


图 16-8 选择枢轴后、放置枢轴后及划分之前的数组

还要注意，第一项至少与枢轴一样小，所以它应该放到较小部分子数组中。所以，我们可以将第一项放在原地，划分算法中从左开始的查找开始于下标  $\text{first}+1$  处。图 16-8b 显示恰在划分前那一刻的数组。

这个机制使得进行两个查找的循环简单了。从左开始的查找查看大于等于枢轴的项。这个查找将会停止，因为最差情形下它会停止在枢轴处。从右开始的查找查看小于等于枢轴的项。这个查找将会停止，因为最差情形下它会停止在第一项。所以循环不需要为阻止查找越出数组边界而做什么特殊的事情。

查找循环停止后，必须将枢轴放到较小部分子数组和较大部分子数组的中间。通过交换  $a[\text{indexFromLeft}]$  和  $a[\text{last}-1]$  处的项可做到这一点。

 **注：**快速排序在划分过程中重排数组中的项。每次划分都将一个项——枢轴——放到其正确的有序位置。两个子数组中位于枢轴之前和之后的项仍位于各自的子数组中。

## 实现快速排序

16.16

**枢轴的选择。**三元中值枢轴选择要求我们排序 3 个项：数组的第一项、中间项及最后一项。我们可以用下列私有方法，通过简单的比较及交换来完成这个任务。

```
// Sorts the first, middle, and last entries of an array into ascending order.
private static <T extends Comparable<? super T>>
 void sortFirstMiddleLast(T[] a, int first, int mid, int last)
```

将数组的第一项、中间项及最后一项的下标传给这个方法，则枢轴将在中间下标处。

**划分。**三元中值枢轴选择假定数组至少有 3 个项。如果你仅有 3 个项，则枢轴的选择过程就是排序它们的过程，所以不再需要划分方法或是快速排序。所以下列划分算法假定，数组中至少含有 4 个项：

```
Algorithm partition(a, first, last)
// Partitions an array a[first..last] as part of quick sort into two subarrays named
// Smaller and Larger that are separated by a single entry—the pivot—named pivotValue .
// Entries in Smaller are <= pivotValue and appear before pivotValue in the array .
// Entries in Larger are >= pivotValue and appear after pivotValue in the array .
// Precondition: first >= 0; first < a.length; last - first >= 3; last <
a.length.
// Returns the index of the pivot.

mid = 数组中间项的下标
sortFirstMiddleLast(a, first, mid, last)
// Assertion: a[mid] is the pivot, that is, pivotValue;
// a[first] <= pivotValue and a[last] >= pivotValue, so do not compare these two
// array entries with pivotValue.

// Move pivotValue to next-to-last position in array
交换 a[mid] 和 a[last - 1]
pivotIndex = last - 1
pivotValue = a[pivotIndex]

// Determine two subarrays:
// Smaller = a[first..endSmaller] and
// Larger = a[endSmaller+1..last-1]
// such that entries in Smaller are <= pivotValue and
// entries in Larger are >= pivotValue.
// Initially, these subarrays are empty .
indexFromLeft = first + 1
indexFromRight = last - 2
done = false
while (!done)
{
 // Starting at index FromLeft, leave entries that are < pivotValue and
 // locate the first entry that is >= pivotValue. You will find one, since the last
 // entry is >= pivotValue.
 while (a[indexFromLeft] < pivotValue)
 indexFromLeft++
 // Starting at index FromRight, leave entries that are > pivotValue and
 // locate the first entry that is <= pivotValue. You will find one, since the first
 // entry is <= pivotValue.
 while (a[indexFromRight] > pivotValue)
 indexFromRight--
 // Assertion: a[indexFromLeft] >= pivotValue and
 // a[indexFromRight] <= pivotValue
 if (indexFromLeft < indexFromRight)
 {
 交换 a[indexFromLeft] 和 a[indexFromRight]
 indexFromLeft++
 indexFromRight--
 }
 else
 done = true
}
// Place pivotValue between the subarrays Smaller and Larger
交换 a[pivotIndex] 和 a[indexFromLeft]
pivotIndex = indexFromLeft
```

16.17

```
// Assertion: Smaller = a[first..pivotIndex-1]
// pivotValue = a[pivotIndex]
// Larger = a[pivotIndex+1..last]
return pivotIndex
```

16.18

**快速排序方法。**在完成快速排序的 Java 代码之前，需要考虑小数组。你已经看到，在调用划分方法之前，数组中至少应该含有 4 个项。但是只允许对大数组使用快速排序还是不够的。段 16.10 给出的快速排序的伪代码显示，即使是对非常大的数组进行划分，最终也会导致递归调用时涉及仅有 2 项的小数组。快速排序的代码必须筛选出这些小数组，并使用其他的方法来排序它们。对于小数组，插入排序是个好选择。事实上，对于含 10 个项的数组，使用插入排序替代快速排序都是合理的。基于这些观察结果，快速排序的实现如下。方法假设了一个常数 MIN\_SIZE，它规定了使用快速排序的最小数组的大小。

```
/** Sorts an array into ascending order. Uses quick sort with
 * median-of-three pivot selection for arrays of at least
 * MIN_SIZE entries, and uses insertion sort for smaller arrays. */
public static <T extends Comparable<? super T>
 void quickSort(T[] a, int first, int last)
{
 if (last - first + 1 < MIN_SIZE)
 {
 insertionSort(a, first, last);
 }
 else
 {
 // Create the partition: Smaller | Pivot | Larger
 int pivotIndex = partition(a, first, last);

 // Sort subarrays Smaller and Larger
 quickSort(a, first, pivotIndex - 1);
 quickSort(a, pivotIndex + 1, last);
 } // end if
} // end quickSort
```



**学习问题 3** 使用方法 quickSort 对数组 9 6 2 4 8 7 5 3 进行升序排序，跟踪排序步骤。假定 MIN\_SIZE 是 4。

## Java 类库中的快速排序

16.19

包 `java.util` 中的 `Arrays` 类使用快速排序对基本类型的数组进行升序排序。方法

```
public static void sort(type[] a)
```

对整个数组 `a` 进行排序，而方法

```
public static void sort(type[] a, int first, int after)
```

对从 `a[first]` 到 `a[after-1]` 的项进行排序。注意，`type` 可以是 `byte`、`char`、`double`、`float`、`int`、`long` 或是 `short` 类型。

## 基数排序

16.20

到目前为止你见过的排序算法对于可比较的对象进行排序。基数排序（radix sort）不使用比较，但为了能进行排序，它必须限定要排序的数据。即它将数组项看作有相同长度的字符串。对于这些受限的数据，基数排序是  $O(n)$  的，故它快于本章介绍的任一个排序方法。但是，它不适合作为通用的排序算法。

我们来看一个例子，对下列 3 位正整数进行基数排序：

123 398 210 019 528 003 513 129 220 294

注意到，19 和 3 都用 0 填满为 3 位整数。基数排序的开始点是根据最右侧的数字对整数进行分组。因为数字可能是 10 个值之一，故我们需要 10 个组，或称为桶 (bucket)。如果桶  $d$  对应于数字  $d$ ，则我们将 123 放到桶 3 中，将 398 放到桶 8 中，以此类推。图 16-9a 所示为这个过程的结果。注意到，每个桶中必须保持整数被接收时的顺序。

依次查看各个桶，则整数按以下的次序排列：

210 220 123 003 513 294 398 528 019 129

将这些整数从桶中移回原来的数组中。然后将它们按中间数字分组，使用现在为空的桶。所以将 210 放到桶 1 中，将 220 放到桶 2 中，将 123 放到桶 2 中，以此类推。图 16-9b 显示这趟扫描的结果。

现在各桶中的整数的次序如下：

003 210 513 019 220 123 528 129 294 398

将这些整数从桶中移回数组，再按最左边的数字进行分组。所以将 003 放入桶 0 中，将 210 放入桶 2 中，将 513 放入桶 5 中，以此类推。图 16-9c 所示为这趟扫描的结果。

现在各桶中的整数是其最终的有序次序：

003 019 123 129 210 220 294 398 513 528

### 旁白：基数排序的起源

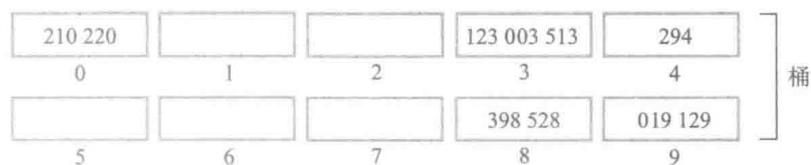
计算机早期阶段，数据存储在穿孔卡片上。每张卡片有 80 列，可以存储 80 个字符。每一列有 12 行，这是可能打孔的位置。称为卡片分类机的机器，根据机器操作员选定的列上打孔的行，将这些卡片分到 12 个仓中。这些仓类似于基数排序中的桶。卡片分类机分类一堆卡片后，操作员每次收集一个仓内的卡片形成新的一堆。分类机根据下一列的孔再次对卡片进行分类。重复这个过程，操作员可对卡片进行排序。

a) 将原始数组分配到各桶中

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 123 | 398 | 210 | 019 | 528 | 003 | 513 | 129 | 220 | 294 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

无序数组

根据最右边的数字将整数分配到桶中



b) 将重排后的数组分配到各桶中

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 210 | 220 | 123 | 003 | 513 | 294 | 398 | 528 | 019 | 129 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

重排后的数组

根据中间的数字将整数分配到桶中

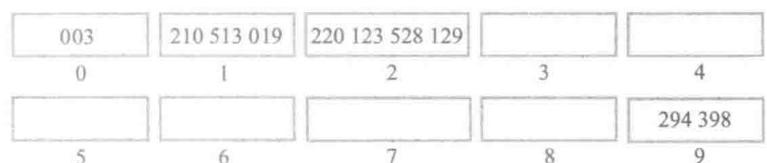


图 16-9 基数排序的步骤

c) 将重排后的数组分配到各桶中

|     |     |     |     |     |     |     |     |     |     |        |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--------|
| 003 | 210 | 513 | 019 | 220 | 123 | 528 | 129 | 294 | 398 | 重排后的数组 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--------|

根据其最左边的数字将整数分配到桶中

|         |         |             |     |   |
|---------|---------|-------------|-----|---|
| 003 019 | 123 129 | 210 220 294 | 398 |   |
| 0       | 1       | 2           | 3   | 4 |
| 513 528 |         |             |     |   |
| 5       | 6       | 7           | 8   | 9 |

d) 排序完成

|     |     |     |     |     |     |     |     |     |     |      |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 003 | 019 | 123 | 129 | 210 | 220 | 294 | 398 | 513 | 528 | 有序数组 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|

图 16-9 (续)

### 基数排序的伪代码

16.21

前面对基数排序的描述假定，要排序的每个整数都含有相同的位数。实际上，这个要求不是必需的，当你要找的数位不存在时可以用 0 表示。例如，如果要找两位整数的百位数字时，应该得到 0。

下列算法描述了对正的十进制整数数组的基数排序。我们从 0 开始对每个整数从右到左的各位进行编号。所以，个位数字是数字 0，十位数字是数字 1，以此类推。

```

Algorithm radixSort(a, first, last, maxDigits)
 // Sorts the array a[first..last] of positive decimal integers into ascending order.
 // maxDigits is the number of digits in the longest integer.

 for (i = 0 to maxDigits - 1)
 {
 清空 bucket[0], bucket[1], ..., bucket[9]
 for (index = first 到 last)
 {
 digit = digit i of a[index]
 将 a[index] 放到 bucket[digit] 的最后
 }
 将 bucket[0], bucket[1], ..., bucket[9] 放回数组 a 中
 }

```

这个算法用到了桶的数组。没有规范说明桶的特性，不过桶可以是 ADT 队列的实例。



学习问题 4 使用算法 `radixSort` 对数组 6340 1234 291 3 6325 68 5227 1638 进行升序排序，跟踪排序步骤。

### 基数排序的效率

16.22

如果数组含有  $n$  个整数，则前一个算法中的内层循环迭代  $n$  次。如果每个整数含有  $d$  位，则外层循环迭代  $d$  次。所以，基数排序是  $O(d \times n)$  的。表达式中的  $d$  告诉我们，基数排序的实际运行时间依赖于整数的大小。但在计算机中，一般的整数的大小限制为 10 位十进制数，或 32 个二进制位。当  $d$  固定且远小于  $n$  时，基数排序仅仅是  $O(n)$  的算法。



注：虽然基数排序对某些数据是  $O(n)$  的算法，但它不适用于所有数据。



**学习问题 5** 基数排序的困难之一是，桶的个数依赖于你要排序的字符串的类型。可以看到，整数排序需要 10 个桶；单词的排序至少需要 26 个桶。如果使用基数排序对单词的数组按字母序排序，则所给的算法中必须进行哪些修改？

## 算法比较

图 16-10 总结了本章及第 15 章给出的排序算法的效率。**16.23** 虽然基数排序是最快的，但它并不总能使用。一般来说，归并排序和快速排序比其他算法要快。

为了说明问题规模对时间效率的影响，对图 16-10 中出现的 4 个增长率函数，将几个  $n$  值的计算结果列在图 16-11 中。当  $n$  是 10 时肯定能使用  $O(n^2)$  的排序算法。当  $n$  是 100 时，希尔排序的平均情况几乎和快速排序一样快。但当  $n$  是 100 万时，平均情况的快速排序比希尔排序要快得多，比插入排序要快得多得多。

|      | 平均情况          | 最优情况          | 最差情况                    |
|------|---------------|---------------|-------------------------|
| 基数排序 | $O(n)$        | $O(n)$        | $O(n)$                  |
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$           |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$                |
| 希尔排序 | $O(n^{1.5})$  | $O(n)$        | $O(n^2)$ 或 $O(n^{1.5})$ |
| 插入排序 | $O(n^2)$      | $O(n)$        | $O(n^2)$                |
| 选择排序 | $O(n^2)$      | $O(n^2)$      | $O(n^2)$                |

图 16-10 使用大  $O$  符号表示的不同排序算法的时间效率

如果数组含有相对较少的项，或是如果它接近有序，则插入排序是好的选择。另外，一般来讲快速排序更可取。注意，当数据集合太大了，不能全部放到内存而必须使用外部文件时，可以使用归并排序。

我们将在第 27 章讨论另一个排序算法，堆排序。这个技术也是  $O(n \log n)$  的，但快速排序通常更可取。

| $f(n)$       | 10     | $10^2$ | $10^3$ | $10^4$  | $10^5$     | $10^6$     |
|--------------|--------|--------|--------|---------|------------|------------|
| $n$          | 10     | 100    | 1000   | 10 000  | 100 000    | 1 000 000  |
| $n \log_2 n$ | 33     | 664    | 9966   | 132 877 | 1 660 964  | 19 931 569 |
| $n^{1.5}$    | 33     | $10^3$ | 31 623 | $10^6$  | 31 622 777 | $10^9$     |
| $n^2$        | $10^2$ | $10^4$ | $10^6$ | $10^8$  | $10^{10}$  | $10^{12}$  |

图 16-11 当  $n$  增大时，增长率函数的比较

## 本章小结

- 归并排序是分治法算法，它将数组一分为二，递归地排序两半，然后将它们合并为一个有序数组。
- 归并排序是  $O(n \log n)$  的。但是它需要用到额外的内存来完成合并过程。
- 快速排序是另一个分治法算法，它由一个项——枢轴——将数组划分为隔开的两个子数组。枢轴在其正确的有序位置上。一个子数组中的项小于等于枢轴，而第二个

子数组中的项大于等于枢轴。快速排序递归地对两个子数组进行排序。

- 大多数情况下，快速排序是  $O(n \log n)$  的。虽然最差情况下是  $O(n^2)$  的，但通常选择合适的枢轴可以避免这种情况。
- 即使归并排序和快速排序都是  $O(n \log n)$  的算法，但是在实际中，快速排序通常更快，且不需要额外的内存。
- 基数排序将数组项看作有相同长度的字符串。初始时，基数排序根据字符串一端的字符（数字）将项分配到桶中。然后排序收集字符串，并根据下一个位置的字符或数字将它们再次分配到桶中。排序继续这个过程，直到所有的字符位置都处理过为止。
- 基数排序不比较数组项。虽然它是  $O(n)$  的，但它不能对所有类型的数据进行排序。所以它不能用作通用的排序算法。

## 练习

1. 假定 80 90 70 85 60 40 50 95 表示 Integer 对象数组。给出使用归并排序对这个数组进行排序的步骤。
2. 考虑段 16.18 给出的方法 quickSort，将对象数组使用快速排序按升序排序。假定 80 90 70 85 60 40 50 95 表示 Integer 对象数组。
  - a. quickSort 第一次划分后数组的内容是什么？（显示所有中间结果。）
  - b. 这次划分过程需要多少次比较？
  - c. 现在枢轴位于称作较小部分和较大部分的两个子数组中间。这个特殊项的位置在后续的排序过程中会改变吗？为什么改变或是为什么不会改变？
  - d. 接下来对 quickSort 的递归调用是什么？
3. 考虑归并排序的合并步骤。
  - a. 合并两个长度各为  $n/2$  的子数组，所需的最少比较次数是多少？
  - b. 给出最优情况下，计算比较次数的递推关系。
  - c. 有根据地猜测下递推关系的解。
4. 当使用三元中值枢轴选择法进行划分时，最差情况下快速排序需要比较多少次？最差情况的大  $O$  是多少？
5. 给出基数排序对下列 Integer 对象数组进行排序的步骤：

783 99 472 182 264 543 356 295 692 491 94

6. 给出基数排序对下列字符串数组按字典序进行排序的步骤：

joke book back dig desk word fish ward dish wit deed fast dog bend

7. 描述扑克牌玩家如何使用基数排序对一手牌进行排序。
8. 考虑由结点链表表示的 Comparable 对象集合。假定你要为这个集合提供一个排序操作。
  - a. 实现一个私有方法，将两个有序链表合并为一个新的有序链表。
  - b. a 中描述的方法可能是有序链表归并排序的一部分。描述如何实现这样一个排序。
9. 回忆一下，如果排序算法没有改变相等对象的相对次序，则它是稳定的。本章和第 15 章的哪些排序算法是稳定的？
10. 段 16.7 显示，你可以通过求解下列递推关系，计算归并排序的效率。

$$t(n) = 2 \times t(n/2) + n \quad \text{当 } n > 1 \text{ 时}$$

$$t(1) = 0$$

通过归纳法证明  $t(n) = n \log_2 n$ 。

11. 与快速排序相比，第 15 章项目 6 中描述的计数排序的效率如何？
12. 段 16.21 提供了对数组进行基数排序的伪代码。实际上，那个算法中的每个桶是一个队列。描述为什么你可以将队列而不是栈用于基数排序。

## 项目

1. 实现递归的归并排序算法。
2. 段 16.8 介绍了迭代的归并排序。本项目继续讨论，给出详细的合并步骤。
  - a. 如果  $n$  是 2 的幂次，如图 16-3 那样，应该从数组头开始，合并一对对单个的项。然后返回到数组头，合并一对对两个项的子数组。最后，应该合并一对对 4 个项的子数组。注意到，每对子数组中的子数组都含有相同个数的项。
  - 一般地， $n$  或许不是 2 的幂次。合并了某些对子数组后，剩下的项太少，不能组成完整的子数组对。图 16-12a 中，合并了单个项的子数组对后，只剩下一个项。然后合并一对两个项的子数组，并合并剩下的两个项的子数组和剩下的一个项的子数组。图 16-12b 和图 16-12c 显示了两种其他的可能。
  - 实现迭代的归并排序。使用段 16.3 给出的算法 `merge`。可使用一个私有方法，它使用 `merge` 合并子数组对。方法完成任务后，可以处理我们刚描述的剩余项。
  - b. 合并两个子数组需要一个额外的临时数组。虽然你必须使用这个额外空间，但能节省下前面的合并算法中将项从临时数组拷贝回原数组所花的时间。如果  $a$  是原始数组，而  $t$  是临时数组，则先将  $a$  的子数组合并到数组  $t$  中。然后，不是将  $t$  拷贝回  $a$  并继续合并，而是判定  $t$  中的子数组，并将它们合并到  $a$  中。如果能这样做偶数次，则不再需要额外的拷贝操作。据此修改 a 中所实现的迭代归并排序。

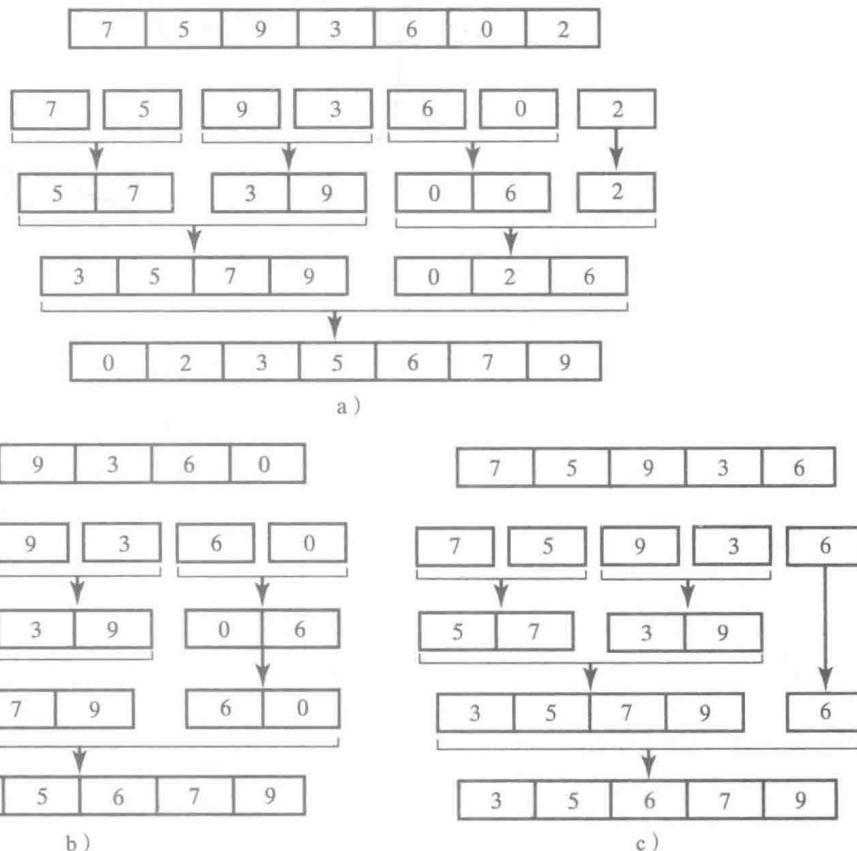


图 16-12 迭代的归并排序中，合并了一个项的子数组后的几种特殊情形

3. 写一个原地的归并排序算法，合并两段时不使用临时数组。你给出的方案的效率如何？
4. 考虑迭代归并排序的下列实现。从开头扫描数组，按照每个有序部分划分段。当找到每个段时，使用下标对来表示，将这些对放到初始为空的向量末尾。

接下来，从向量中删除前两个对，并合并它们所表示的数据段。注意，这些段在数组中是相邻的。合并得到一个更大的有序段。将表示结果段的下标对放到向量末尾。重复本段所说的步骤，直到向量中仅有一个项时为止。

有时，处理过程中，向量开头的两个下标对可能表示不相邻的两段。这种情形下，将第一个下标对移到向量的末尾，并继续进行处理。

- a. 这个算法的最优性能是多少？
- b. 这个算法的最差性能是多少？
- c. 实现算法。
5. 前一个项目中将向量用作队列。重做这个项目，但使用队列而不是使用向量完成。
6. 实现划分方法，从而实现快速排序。
7. 修改快速排序的实现如下。如果数组有 7 个项，则选择中间的项作为枢轴。对于含 8 ~ 40 个项的数组，使用段 16.14 和段 16.16 描述的三元中值枢轴选择法。对于更大的数组，枢轴是差不多等距的 9 个项的中值，9 个项中包括第一项、最后一项及中间项在内。对于少于 7 个项的数组，使用插入排序替代快速排序。
8. 扩展第 15 章的项目 1，给出本章介绍的归并排序和快速排序算法的图示化说明。
9. 统计学家常常感兴趣数据集合的中位数。集合中，大于中位数的值的个数，与小于中位数的值的个数差不多相等。找到中位数的一个办法是对数据进行排序，并取位于——或接近——集合中间的值。但排序要做的事情，相对于找中位数来说，是杀鸡用牛刀。对于一个合适的  $k$  值，你只需找到集合中第  $k$  小的项。要找到  $n$  个项的中位数，可以取  $k$  为  $n/2$  的取整，即  $[n/2]$ 。

可以使用快速排序的划分策略来找到数组中第  $k$  小的项。如图 16-5 所示，选择了枢轴并形成较小部分和较大部分的子数组后，可以得到下列结论之一：

- 如果较小部分含有  $k$  个或更多个项，则它必定含有第  $k$  小的项。
- 如果较小部分含有  $k-1$  个项，则第  $k$  小的项就是枢轴。
- 如果较小部分含有少于  $k-1$  个项，则第  $k$  小的项必定在较大部分中。

现在可以实现找到第  $k$  小项的递归方法。第一个和最后一个结论对应于递归调用。另一个是基础情形。

实现找到无序数组中第  $k$  小项的递归方法。使用你的方法查找数组的中位数。

10. 二进制基数排序将含  $n$  个整数值的数组  $a$ ，根据它们的二进制位而不是它们的十进制位进行排序。这个排序仅需要两个桶。将桶表示为  $2 \times n$  的数组。不需要在每趟扫描结束时将桶的内容拷贝回数组  $a$ 。而只是将第二个桶的值加到第一个桶的最后即可。
- 实现这个算法。
11. 实现段 16.21 给出的基数排序，使用队列表示每个桶。
12. 重做第 15 章的项目 3，但对归并排序和快速排序进行比较。
13. 对结点链表中的对象实现归并排序。比较归并排序的这个版本的运行时间，及相同的数据放在数组中使用快速排序进行排序的运行时间。参见第 4 章结尾处描述的如何对一段 Java 代码进行计时的项目。
14. 实现对结点链表中字符串的基数排序。
15. 第 14 章项目 9 要求你设计并实现一个使用递归和回溯的排序算法。
  - a. 将你给出的排序算法的时间效率，与本章的算法的效率进行比较。
  - b. 根据本章学到的内容，如何改进你的排序算法？

# 可变及不可变对象

**先修章节：**附录 C、第 10 章

当类有公有赋值方法或设置方法时，客户可以使用这些方法来改变类的对象。虽然这个功能似乎很合理，但如果另一个类用特殊的方式组织这些对象，它就不合理了。例如，第 17 章描述的 ADT 有序表，它根据你的规范——例如字母序——保持项的有序性。如果客户想改变有序表中的一个名字，这可能会破坏表的有序性。

本插曲来研究防止这个问题的办法。它只要求客户将不能改变的对象放到一个集合中。Java 插曲 9 将提出第二个策略，它需要集合复制或克隆（clone）客户添加的任意对象。采用这项技术，客户不能指向集合中的备份，所以不能改变它。

## 可变对象

到目前为止，你学习的许多类都有私有的数据域，及查找或修改这些域的公有方法。J6.1 如你所知，这样的方法称为访问方法和赋值方法——或叫获取方法和设置方法。属于一个有公有赋值方法的类的对象称为可变的（mutable）——如我们在第 7 章提到过的——因为客户可以使用设置方法来改变对象的数据域的值。例如，附录 B 段 B.16 中由两段组成的名字的 Name 类。它有两个数据域：

```
private String first; // First name
private String last; // Last name
```

为能让客户改变这些域的值，我们给类增加赋值方法 `setFirst` 和 `setLast`。要查找这些域，它有访问方法 `getFirst` 和 `getLast`。



**注：**可变对象属于对其数据域有赋值（设置）方法的类。

让我们使用这个类创建 Chris Coffee 对应的对象，写下面的 Java 语句：J6.2

```
Name chris = new Name("Chris", "Coffee");
```

图 JI6-1 说明了这个对象及其引用变量 `chris`。

现在假定，我们创建一个线性表，然后将 `chris` 添加到表中，语句如下：

```
ListInterface<Name> nameList = new LList<>();
nameList.add(1, chris);
```



图 JI6-1 一个对象及它的引用变量 `chris`

因为 `chris` 是可变对象，故我们可以改变它的数据域，例如写如下的语句：

```
chris.setLast("Smith");
```

修改后，对象 `chris` 代表名字 Chris Smith。这个没什么可奇怪的。但出人意料的是线性表已经改变了！是的，如果想获取线性表中的第一项，比如用下面这样的语句：

```
System.out.println(nameList.getEntry(1));
```

则我们将得到 Chris Smith 而不是 Chris Coffee。

J6.3 在修改名字之前所创建的线性表，是如何能含有被修改的名字的？记住，在 Java 中，线性表含有指向客户放置到线性表中的实际对象的引用。所以线性表有一个引用指向它的第一项，客户也有，因为它有变量 chris，如图 JI6-2a 所示。

当执行下列语句修改对象时

```
chris.setLast("Smith");
```

我们改变了对象的一个，且是唯一一个备份，如图 JI6-2b 所示。因为线性表仍指向那个对象，所以 nameList.getEntry(1) 返回指向那个对象的引用。Java 的这个特点能让客户方便地修改线性表中已经放置的对象。

 注：当客户创建一个可变对象且将其添加到 ADT 线性表中时，通常只有一个对象备份存在。所以，如果客户修改了对象，则线性表也改变了。理想的方式是，客户使用 replace 操作来改变线性表中的项，但我们不能强迫客户这样做。

J6.4 改变集合中可变对象的能力，会让客户破坏集合的完整性。例如，假定我们按字典序创建了一个名字列表。如果我们写

```
Name jesse = new Name("Jesse", "Java");
Name rob = new Name("Rob", "Bean");
ListInterface<Name> alphaList = new AList<>();
alphaList.add(jesse);
alphaList.add(1, rob);
```

会得到字典序的线性表

Rob Bean

Jesse Java

现在如果写

```
rob.setLast("Smith");
```

则线性表变为

Rob Smith

Jesse Java

这个线性表不再按字典序有序。解决这个问题的一种办法是，要求客户使用不可变对象，如下一段所描述的。

## 不可变对象

J6.5 如第 7 章所说，不可变对象是其数据域不能被客户改变的对象。不可变对象所属的类没有公有的赋值（设置）方法，所以一旦创建了对象，就不能修改它的数据域。如果需要修改它们，则只能丢掉对象然后再创建一个具有修改值的新对象。这样的类称为只读（read only）。

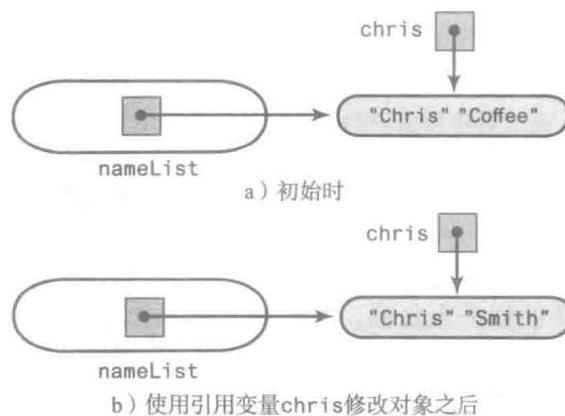


图 JI6-2 线性表 nameList 中的一个对象，在其被修改之前和之后

将不可变对象放到有序表中的客户，不能修改那些对象，所以不会破坏线性表的有序性。

-  注：不可变对象属于只读类。这样的对象可以阻止客户修改其数据域的值。
- 注：当集合按某种方式组织对象时，客户不应该直接修改对象来破坏这个组织方式。然而，如果客户保留了指向对象的引用，那么客户恰恰能做到这一点。只将不可变对象添加到集合中，就能阻止这个问题的发生。

**可变或不可变？**大多数类都有设置方法，所以它们的实例都是可变的。能改变对象的数据是方便且高效的，特别是当对象的状态在程序执行期间必须经常改变时。例如，银行必须定期更新表示你支票账户的对象。如果那个对象是不可变的，则每次有改变时，都要将它丢弃，然后再创建表示更新数据的新对象。但是替换一个对象要比修改它花更长的时间。

另一方面，共享一个可变对象可能是危险的。假定你有两个引用 *x* 和 *y* 指向同一个对象。如果使用 *x* 来修改对象，当再使用 *y* 来引用它时或许会弄糊涂。但共享不可变对象是安全的，因为不管你如何引用它，它们都保持不变。

-  **程序设计技巧：**如果对象要共享，或添加到的集合的完整性会因对象的改变而改变时，那就使用不可变对象。如果数据经常改变则使用可变对象。

## 创建只读类

要将前一个类 `Name` 转为只读类，可以将方法 `setFirst`、`setLast` 和 `setName` 的访问修饰符由 `public` 改为 `private`，以阻止客户调用它们。（我们选择的是全部去掉这些方法，且修改调用它们的其他方法。）还去掉了 `Name` 的方法 `giveLastNameTo`。

现在来看得到的类 `ImmutableName`。如果将 `ImmutableName` 的实例放到线性表或有序表中，则不能使用为它们保留下来的任何的引用来修改这些对象。当然，可以使用 ADT 线性表的 `replace` 操作来替换线性表中的某个项，不过，正如你在第 17 章将看到的，对于有序表没有这样的操作。要改变有序表中的项，必须先删除项然后再添加一个新项。用这个办法，有序表可保持其有序性。

-  **安全说明：**假定 `ImmutableName` 有保护的赋值方法，如 `setFirst` 和 `setLast`。程序员可以使用继承来修改这个类的行为。假定从 `ImmutableName` 派生了一个可变对象的类。然后可以将新类的实例添加到 `ImmutableName` 对象的有序表中。因为这些项都是可变的，故可以修改它们，进而破坏了有序表的次序。为阻止这个事情的发生，可以让类是终极的，参见第 2 章段 2.15 结尾处的安全说明中所描述的内容。

程序清单 JI6-1 将 `ImmutableName` 定义为终极类。因为类没有设置方法，所以我们没有定义默认的构造方法。你可以定义一个，尽管它没什么用处。

J6.6

### 程序清单 JI6-1 只读类 `ImmutableName`

```

1 public final class ImmutableName
2 {
3 private String first; // First name
4 private String last; // Last name
5 }
```

J6.7

J6.8

```

6 public ImmutableName(String firstName, String lastName)
7 {
8 first = firstName;
9 last = lastName;
10 } // end constructor
11
12 public String getFirst()
13 {
14 return first;
15 } // end getFirst
16
17 public String getLast()
18 {
19 return last;
20 } // end getLast
21
22 public String getName()
23 {
24 return toString();
25 } // end getName
26
27 public String toString()
28 {
29 return first + " " + last;
30 } // end toString
31 } // end ImmutableName

```

J6.9 我们还有一件事情要说明一下。假定有一个类，`Name` 对象是其数据域。这个类有访问方法，包括返回 `Name` 域值的方法，但没有赋值方法。不过，这个类的客户可以访问 `Name` 域，然后使用 `Name` 的设置方法来改变域的值。换句话说，类不是只读的。要让它成为只读的，可以定义数据域为终极的。注意，`ImmutableName` 的域是字符串，它是不可改变的，所以我们不需要让它们是终极的。



### 注：只读类的设计指南

- 类应该是终极的。
- 数据域应该是私有的。
- 可变对象的数据域应该是终极的。
- 类不应该有公有的设置方法。



### 学习问题 1 定义 `ImmutableName` 的一个构造方法，带一个 `Name` 对象的参数。

## 伴生类

J6.10 虽然不可变对象对某些应用是可取的，但可变对象也有用武之地。有时我们还想用不可变及可变这两种形式表示同一个对象。这种情况下，一对伴生类（companion class）可能是方便的。类 `ImmutableName` 和 `Name` 就是这样两个伴生类的例子。两个类中的对象都表示名字，但一种对象不能改变，而另一种对象则可以修改。

要让类更便利一些，可以包含一些将对象从一个类型转为另一个类型的构造方法或方法。例如，可以为类 `ImmutableName` 添加下列构造方法和方法：

```
// Add to the class ImmutableName:
public ImmutableName(Name aName)
{
 first = aName.getFirst();
 last = aName.getLast();
} // end constructor
public Name getMutable()
{
 return new Name(first, last);
} // end getMutable
```

类似地，可以为类 Name 添加下列构造方法和方法：

```
// Add to the class Name
public Name(ImmutableName aName)
{
 first = aName.getFirst();
 last = aName.getLast();
} // end constructor
public ImmutableName getImmutable()
{
 return new ImmutableName(first, last);
} // end getMutable
```

图 JI6-3 说明了两个类 Name 和 ImmutableName。

| Name                                                                                                                                                                      | ImmutableName                                                      |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| first<br>last                                                                                                                                                             | first<br>last                                                      |
| getFirst()<br>getLast()<br>getName()<br>setFirst(firstName)<br>setLast(lastName)<br>setName(firstName, lastName)<br>giveLastNameTo(aName)<br>toString()<br>getImmutable() | getFirst()<br>getLast()<br>getName()<br>toString()<br>getMutable() |

图 JI6-3 类 Name 和 ImmutableName



示例：我们来看看如何使用前面在伴生类中所添加的方法。如果使用下列语句定义一个 Name 对象 flexibleName J6.11

```
Name flexibleName = new Name("Maria", "Mocha");
```

且不需要再改变它，则可以使用 ImmutableName 的构造方法，如下所示：

```
ImmutableName fixedName = new ImmutableName(flexibleName);
```

新对象 fixedName 与 flexibleName 有相同的数据域，但它是不可变的。另外，可以调用 Name 的 getImmutable 方法，如下所示：

```
ImmutableName fixedName = flexibleName.getImmutable();
```

类似地，如果有 ImmutableName 的另一个实例，例如

```
ImmutableName persistent = new ImmutableName("Jesse", "Java");
```

且我们还需要修改它，则可以定义一个新的可变对象

```
Name transient = new Name(persistent);
```

或是

```
Name transient = persistent.getMutable();
```

新对象 `transient` 与 `persistent` 有相同的数据域，不过它还有修改数据域的设置方法。



**注：**如果不希望任何人将类 `Name` 用作基类，可以将它声明为终极的。



**学习问题 2** 写 Java 语句，执行下列步骤：

- 创建类 `Name` 的一个对象。
- 不改变对象的数据域，将其转换为不可变对象。
- 将不可变对象添加到线性表 `nameList` 中。

**学习问题 3** 写 Java 语句，执行下列步骤：

- 创建类 `ImmutableName` 的一个对象。
- 不改变对象的数据域，将其转换为可变对象。
- 修改新对象的姓。
- 将改变后的可变对象转换为不可变对象。



**注：**Java 的 `String` 类是一个只读类。即 `String` 的实例是不可变的。一旦创建了一个字符串，就不能修改它。但通常，字符串应用中需要你删除字符串的一部分，或是将两个字符串连接起来。对于这样的应用，Java 提供了可变字符串的类 `StringBuilder`。`StringBuilder` 提供了添加、删除或替换子串的几个方法，从而能修改字符串。补充材料 1（在线）中描述了属于这两个类的一些方法。

`String` 和 `StringBuilder` 是一对伴生类。`String` 有一个构造方法，它带一个 `StringBuilder` 实例作为实参，并得到一个有同样值的不可变字符串。`StringBuilder` 有一个类似的构造方法，能从不可变字符串创建可变字符串。`StringBuilder` 还有返回 `String` 实例的方法 `substring` 和 `toString`。

# 有 序 表

先修章节：第 4 章、第 9 章、第 10 章、第 12 章、Java 插曲 6

## 目标

学习完本章后，应该能够

- 在程序中使用有序表
- 描述 ADT 线性表和 ADT 有序表之间的不同
- 使用结点链表实现 ADT 有序表
- 使用 ADT 线性表的操作实现 ADT 有序表

第 10 章介绍了 ADT 线性表。线性表中的项仅按其在线性表中的位置有序。所以线性表有第一项、第二项，等等。这个 ADT 能让你根据所想要的任何准则——例如字典序或是时间序——对项进行排列。实际上，第 10 章展示了一个示例，使用线性表将名字按字典序进行组织。为此，必须由客户来决定某个项在表中的具体位置。

假定某应用创建了一个线性表，然后在某些时刻必须对表中的项按数值或字典序进行排序，比如你可能在 ADT 线性表中增加一个排序操作。能够使用第 15 章和第 16 章中所给的一个算法实现这个操作。不过当你的应用仅需要有序数据时，对你来说，拥有一个数据有序的 ADT 比拥有 ADT 线性表更方便。有序表就是这样的一个 ADT。

当从有序表中添加或删除项时，仅需提供项本身。不用指明项应在线性表中的什么位置。ADT 可以为你判定它的位置。

本章介绍 ADT 有序表的操作，提供使用有序表的示例，介绍 Java 的两种实现方式。有一种实现方式是使用 ADT 线性表，但它不是特别有效的。第 18 章将介绍类的重用，在讨论继承的使用时，也介绍有序表更高效的一种实现方式。

## ADT 有序表的规范说明

ADT 线性表将给定集合中对象的组织方式交由客户指定。客户可以按其所需要的任何次序维护对象。假定你想要一个按字典序排列的名字或其他字符串的线性表。当然可以使用 ADT 线性表来完成这个任务，但必须规范说明每个字符串在线性表中应处的位置。如果线性表本身在你添加项时已按字典序排列了，这样会不会更方便？你所需的是一个不同的 ADT，名为有序表（sorted list）。

回忆一下，要使用 ADT 线性表的 add 操作，需要指明新项，然后必须指明它在线性表中的位置。这样的操作不是 ADT 有序表所希望的，因为有序表自己负责组织这些项。如果允许你指定新项的位置，可能就破坏了有序表中项的次序。而 ADT 有序表的 add 操作仅需要新项。将新项与有序表中的其他项进行比较，来确定新项的位置。所以，有序表中的项必须是能相互比较的对象。

那么，能将什么样的对象放到有序表中呢？字符串是一种可能，因为 String 类提供了用于比较两个字符串的 compareTo 方法。一般地，可以得到的有序表是由拥有 compareTo

方法的类的任何对象组成的。正如你在 Java 插曲 5 的开头部分所看到的，这样的类实现了接口 Comparable。因为 Java 的包装类，例如 Integer 和 Double，实现了 Comparable 接口，所以你可以将它们的实例放到有序表中。

**17.2** 现在来看这个 ADT 可能具有的操作。为简单起见，我们允许有序表中含有重复项。坚持让有序表含有唯一项的这个条件，有时会令实现更复杂，我们将这个变型留作练习。

我们已经提到，你可以将一个项添加到有序表中。因为有序表自己决定新项的位置，所以可以向 ADT 询问这个位置。即你可以查询已有项的位置，或是若将某个项添加到有序表中时它应处的位置。还可以向 ADT 查询，有序表中是否含有某个项。显然，还应该能删除一项。

现在更详细地规范说明这些操作。

#### 抽象数据类型：有序表

##### DATA

- 按值有序排列并有相同数据类型的对象的集合
- 集合中对象的个数

##### 操作

| 伪代码                               | UML                                            | 描述                                                                                                               |
|-----------------------------------|------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>add(newEntry)</code>        | <code>+add(newEntry: T): void</code>           | 任务：将 newEntry 添加到有序表中，且有序表仍保持有序<br>输入：newEntry 是要添加的对象<br>输出：无                                                   |
| <code>remove(anEntry)</code>      | <code>+remove(anEntry: T): boolean</code>      | 任务：从有序表中删除 anEntry 第一次或唯一一次的出现<br>输入：anEntry 是要删除的对象<br>输出：如果在有序表中找到 anEntry 并删除，则返回真，否则返回假。后一种情况下，有序表维持不变       |
| <code>getPosition(anEntry)</code> | <code>+getPosition(anEntry: T): integer</code> | 任务：得到 anEntry 第一次或唯一一次出现的位置<br>输入：anEntry 是要找的对象<br>输出：如果在有序表中找到 anEntry，则返回它的位置。否则返回 anEntry 应该在有序表中的位置，但以负整数表示 |

下列操作的行为与在第 10 章中已经描述过的 ADT 线性表中的一样：

```
getEntry(givenPosition)
contains(anEntry)
remove(givenPosition)
clear()
getLength()
isEmpty()
toArray()
```

**17.3** 前两个方法很简单，但 getPosition 值得讨论一下。给定有序表中的一个项，方法 getPosition 如你所愿返回项在有序表中的位置。和对 ADT 线性表的处理一样，我们从 1 开始为项编号。但是，如果给定的项不在有序表中时会怎么样呢？这种情况下，getPosition 返回项应该在有序表中的位置值。但是返回值是一个负数，用来表明项不在

有序表中。例如，如果 `missingObject` 不在有序表 `sList` 中，但应该在位置 3，则 `sList.getPosition(missingObject)` 应该返回 -3。注意，`getPosition` 对空表返回 -1，表示给定的项应该在位置 1 处。

有序表还具有 ADT 线性表中的某些操作，但不是全部的。我们已经提到过，将一个项添加在给定位置的操作是不可能的，因为如果那样，客户可能会破坏有序表的次序。出于同样的原因，线性表的 `replace` 方法也不能用于有序表。但 ADT 线性表的其他操作可用于有序表，包括获取或删除给定位置的项。方法 `getEntry` 和 `remove` 都有一个位置作为形参，不过它们不改变有序表中项的相对次序。

虽然线性表的 `remove` 方法返回从线性表中删除的对象，但对于有序表的 `remove` 方法，并不一定这样做。客户至少已经有这个项的一个备份后，才能用它来调用有序表的 `remove` 方法。

程序清单 17-1 中的 Java 接口更详细地规范说明了这些操作。Java 插曲 5 的段 J5.13 中介绍的符号 `? super T`，表明是泛型 `T` 的超类。17.4

### 程序清单 17-1 SortedListInterface 接口

```

1 /** An interface for the ADT sorted list.
2 * Entries in the list have positions that begin with 1.
3 */
4 public interface SortedListInterface<T extends Comparable<? super T>>
5 {
6 /** Adds a new entry to this sorted list in its proper order.
7 * The list's size is increased by 1.
8 * @param newEntry The object to be added as a new entry. */
9 public void add(T newEntry);
10
11 /** Removes the first or only occurrence of a specified entry
12 from this sorted list.
13 * @param anEntry The object to be removed.
14 * @return True if anEntry was located and removed; */
15 * otherwise returns false. */
16 public boolean remove(T anEntry);
17
18 /** Gets the position of an entry in this sorted list.
19 * @param anEntry The object to be found.
20 * @return The position of the first or only occurrence of anEntry
21 * if it occurs in the list; otherwise returns the position
22 * where anEntry would occur in the list, but as a negative
23 * integer. */
24 public int getPosition(T anEntry);
25
26 // The following methods are described in Segment 10.9 of Chapter 10
27 // as part of the ADT list:
28
29 public T getEntry(int givenPosition);
30 public boolean contains(T anEntry);
31 public T remove(int givenPosition);
32 public void clear();
33 public int getLength();
34 public boolean isEmpty();
35 public T[] toArray();
36 } // end SortedListInterface

```

 注：ADT 有序表可以添加、删除或查找一个项，给定的项作为一个实参。有序表有几个与 ADT 线性表一样的操作，名字是 `getEntry`、`contains`、`remove`（按位置）、

`clear`、`getLength`、`isEmpty` 和 `toArray`。但是，有序表不会让你按位置添加或替换一个项。

 **注：**ADT 有序表通过比较表中的项，来决定它们的次序。所以，依第7章段7.19中的设计决策所讨论的，有序表中不能含有 null 项。

 **注：升序还是降序？**

ADT 有序表的规范说明要求，项按适当的排序顺序放置。但是规范说明中没有指明这个顺序是升序还是降序。这由有序表的实现者来抉择。在后续的示例中，我们假定有序表按升序组织表中的项。

 **注：方法 `remove(givenPosition)` 与 `remove(anEntry)` 的比较**

ADT 有序表有两个删除项的方法，但这两个方法的功能不同。从有序表中删除给定项的方法，根据成功或失败而返回真或是假。这里，失败仅意味着在有序表中没有找到项。客户可以使用这个方法在有序表中查找一个给定的项，如果找到则删除它。而从有序表中删除给定位置的项的方法，或是返回这个项，或是如果位置不在合理范围内则抛出一个异常。异常是必需的，因为超出范围的位置可能会有更严重的后果，与找不到项的含义不同。

 **注：从有序表中添加或删除项的方法的命名**

从有序表中删除一个给定项的方法，与删除给定位置项的方法有相同的名字：`remove`。为避免这两个方法之间混淆，你可以将前一个方法命名为 `removeEntry`。如果你愿意，可以将方法 `add` 改为 `addEntry`。

## 使用 ADT 有序表

17.5

 **示例。**为了演示前一节中规范说明的 ADT 有序表的操作，我们先创建一个字符串的有序表。从声明并分配线性表 `nameList` 开始，假定 `SortedList` 实现了接口 `SortedListInterface` 中规范说明的 ADT 操作：

```
SortedListInterface<String> nameList = new SortedList<>();
```

接下来，按任意次序添加名字，我们知道 ADT 将按字典序组织它们：

```
nameList.add("Jamie");
nameList.add("Brenda");
nameList.add("Sarah");
nameList.add("Tom");
nameList.add("Carlos");
```

现在有序表中含有下列项：

Brenda

Carlos

Jamie

Sarah

Tom

假定有刚刚给出的有序表，下面是有序表上 ADT 操作的几个示例：

|                                            |                                     |
|--------------------------------------------|-------------------------------------|
| <code>nameList.getPosition("Jamie")</code> | 返回 3, <i>Jamie</i> 在有序表中的位置         |
| <code>nameList.contains("Jill")</code>     | 返回假，因为 <i>Jill</i> 不在有序表中           |
| <code>nameList.getPosition("Jill")</code>  | 返回 -4，因为 <i>Jill</i> 应位于有序表的位置 4    |
| <code>nameList.getEntry(2)</code>          | 返回 <i>Carlos</i> ，因为有序表的位置 2 是这个字符串 |

现在使用下列语句删除 Tom 及有序表中的第一个名字：

```
nameList.remove("Tom");
nameList.remove(1);
```

现在有序表中含有

- Carlos
- Jamie
- Sarah

删除最后的项 Tom，不会改变表中其他项的位置，但删除第一项时会改变。Carlos 现在位于位置 1 而不是位置 2。

 **学习问题 1** 假定 wordList 是单词的无序表。使用 ADT 线性表和 ADT 有序表的操作，创建这些单词的有序表。

**学习问题 2** 假定前一问中创建的有序表非空，写 Java 语句，完成

- 显示有序表中的最后一项。
- 不删除有序表的第一项，将它再次添加到有序表中。

## 链式实现

如同所有的 ADT 一样，你可以选择几种不同的方式来实现有序表。例如，可以将有序表的项保存在数组、结点链表、向量实例或是 ADT 线性表的实例中。本章，我们考虑结点链表及 ADT 线性表的实例。在第 18 章，我们将使用继承和 ADT 线性表来开发完全不同的实现。

 **类的框架。** 使用结点链表来保存有序表中的项的实现过程，有几个细节与第 12 章学习的 ADT 线性表的链式实现是一样的。具体来说，它有相同的数据域，类似的构造方法，有几个方法的实现也是相同的，内部类 Node 的定义也一样。将实现 ADT 有序表的类定义概括在程序清单 17-2 中。

### 程序清单 17-2 ADT 有序表链式实现的框架

```
1 public class LinkedSortedList<T extends Comparable<? super T>>
2 implements SortedListInterface<T>
3 {
4 private Node firstNode; // Reference to first node of chain
5 private int numberOfEntries;
6
7 public LinkedSortedList()
8 {
9 firstNode = null;
10 numberOfEntries = 0;
11 } // end default constructor
12
13 < Implementations of the sorted list operations go here. >
```

```

14
15 private class Node
16 {
17 private T data;
18 private Node next;
19 <Constructors>
20
21 < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
22
23 } // end Node
24 } // end LinkedList

```

## 方法 add

17.8

**找到插入点。**将项添加到有序表中，需要找到新项在有序表中的位置。因为我们假定项是升序排列的，所以将新项与有序表中的各项进行比较，直到到达一个不小于新项的项。图 17-1 描述了结点链表，每个结点中含有一个名字，且按字典序排序。这个图表表明，要添加的名字 Ally、Cathy、Luke、Sue 和 Tom 应该在链表中的插入位置，及到达这些位置时发生的最后一次比较。

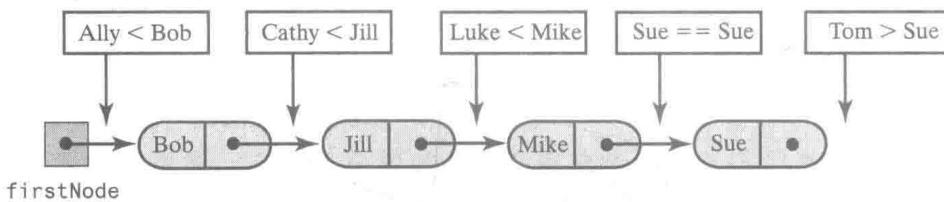


图 17-1 将追加的名字插入有序结点链表中的位置

从图中可以看到，在字符串比较中，Ally 小于 Bob，故它应该插入在链表的开头。要想知道 Luke 的插入位置，应该发现 Luke 大于 Bob 和 Jill，但小于 Mike。所以 Luke 应该位于链表中 Mike 的前面。而 Sue 已经保存在一个结点中了。应该发现 Sue 大于 Bob、Jill 和 Mike，但不大于 Sue。所以应该将新项 Sue 就插入在已有项 Sue 的前面。最后，Tom 大于链表中当前的所有项，所以应该将它添加到链表表尾。



注：给定一个其项按升序排列的有序表，新项就插入第一个不小于新项的项的前面。

17.9

**算法。**回忆第 12 章段 12.10，将新结点添加在链表表头，不同于将它插入链表的其他位置。在开头的添加是简单的，因为 `firstNode` 指向链表的首结点。要添加在其他位置时，需要一个引用，来指向新结点最终所处位置的前一个结点。所以，当遍历结点链表查找新项应处位置时，必须维护一个指向当前结点的前一个结点的引用。

描述这个有序表策略的高级算法如下所示。

**Algorithm** add(newEntry)

// Adds a new entry to the sorted list.

分配一个新结点，其中含有 `newEntry` 值

查找链表，直到找到含有 `newEntry` 的结点，或是越过了这个值应该在的位置

让 `nodeBefore` 指向插入点的前一个结点

**if** (链表是空的，或者新结点位于链表表头)

    将新结点添加在链表表头

**else**  
将新结点添加在nodeBefore指向的结点的后面  
有序表的长度加1

add 的迭代实现。前面这个算法的 Java 实现如下所示。使用私有方法 getNodeBefore [17.10] 在链表中查找插入点的前一个结点。

```
public void add(T newEntry)
{
 Node newNode = new Node(newEntry);
 Node nodeBefore = getNodeBefore(newEntry);
 if (isEmpty() || (nodeBefore == null))
 {
 // Add at beginning
 newNode.setNextNode(firstNode);
 firstNode = newNode;
 }
 else
 {
 // Add after nodeBefore
 Node nodeAfter = nodeBefore.getNextNode();
 newNode.setNextNode(nodeAfter);
 nodeBefore.setNextNode(newNode);
 } // end if
 numberOfEntries++;
} // end add
```

私有方法 getNodeBefore。我们还需要实现私有方法 getNodeBefore。在遍历线性表时需要两个引用。显然，我们需要一个指向当前结点的引用，这样我们可以将当前结点的项与要添加的项进行比较。但我们还必须维护指向前一个结点的引用，因为这正是方法返回的引用。下列实现中，这两个引用分别是 currentNode 和 nodeBefore。 [17.11]

```
// Finds the node that is before the node that should or does
// contain a given entry.
// Returns either a reference to the node that is before the node
// that contains—or should contain—anEntry, or null if no prior node exists
// (that is, if anEntry is or belongs at the beginning of the list).
private Node getNodeBefore(T anEntry)
{
 Node currentNode = firstNode;
 Node nodeBefore = null;
 while ((currentNode != null) &&
 (anEntry.compareTo(currentNode.getData()) > 0))
 {
 nodeBefore = currentNode;
 currentNode = currentNode.getNextNode();
 } // end while
 return nodeBefore;
} // end getNodeBefore
```

回忆一下，方法 compareTo 根据比较结果是小于、等于或大于，分别返回负整数、0 或正整数。



学习问题 3 在方法 getNodeBefore 的 while 语句中，运算符 && 连接的两个逻辑表达式的次序有多重要？解释之。

学习问题 4 如果有序表为空，则 getNodeBefore 返回什么？如何使用这个事实来简化段 17.10 给出的方法 add 的实现？

**学习问题 5** 假定你使用前面的 `add` 方法，将项添加到有序表中。如果项已经在表中，则 `add` 将它添加在有序表的什么位置：在项的第一次出现之前？在项的第一次出现之后？在项的最后一次出现之后？或是其他什么位置？

**学习问题 6** 如果将 `getNodeBefore` 方法的 `while` 语句中的 `>` 改为 `>=`，则学习问题 5 的答案是什么？

17.12

**递归地思考。** 使用递归去处理结点链表，是迭代方法强有力的替代方案。基本概念是简单的，但正如你看到的，实现中要涉及更多的内容，因为 Java 将对象作为引用传递给方法。

回忆第 9 章段 9.20，你可以处理链表的首结点，然后递归地处理链表的其余部分。所以，要将新结点添加到有序链表中，可以使用下面的逻辑：

```
if (链表是空的，或者新结点位于链表表头)
 将新结点添加在链表表头
else
 忽略首结点，将新结点添加在链表的其余部分
```

图 17-2 说明了递归地将 Luke 添加到有序名字链表中所涉及的逻辑。因为 Luke 大于 Bob，可以递归地考虑从 Jill 开始的子链表。Luke 还大于 Jill，所以现在考虑从 Mike 开始的子链表。最后，Luke 小于 Mike，故实际上添加操作位于这个子链表的开头——即在 Mike 的前面。添加到链表或子链表的开头是这个递归的基础情形。高兴的是，链表的开头是处理添加的最容易的位置。

如果 `currentNode` 初始时指向链表，然后指向链表的其余部分，则我们可以细化前一个逻辑，如下所示。

```
if ((currentNode == null) 或 (newEntry <= currentNode.getData()))
 currentNode = new Node(newEntry, currentNode)
else
 递归地将newEntry添加在currentNode.getNextNode()返回的链表表头
```

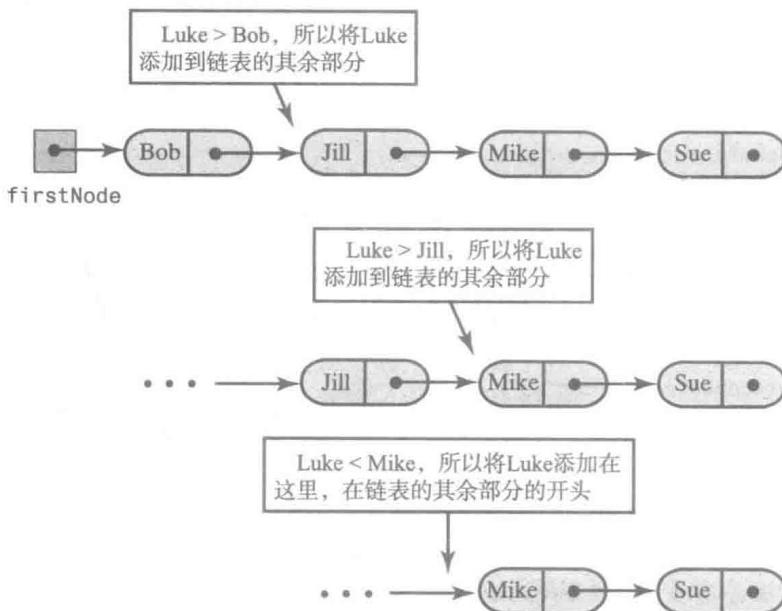


图 17-2 递归地将 Luke 添加到有序的名字链表中

17.13

**add 的递归实现。** 段 9.20 中的示例显示了链表的内容。因为操作没有改变链表，故它

的递归形式是简单的。显然，在目前这个例子中，方法 add 确实要改变链表。让递归方法进行这样的修改，对 Java 来讲是一大挑战。

在我们描述 add 方法为什么能奏效之前，先来看看它的递归实现。在段 9.19 中已经了解到，写一个私有方法去执行递归，然后写一个公有方法——通常是实现 ADT 操作的方法——去调用这个私有方法。所以有下列方法定义：

```
public void add(T newEntry)
{
 firstNode = add(newEntry, firstNode);
 numberOfEntries++;
} // end add

private Node add(T newEntry, Node currentNode)
{
 if ((currentNode == null) ||
 (newEntry.compareTo(currentNode.getData()) <= 0))
 {
 currentNode = new Node(newEntry, currentNode);
 }
 else
 {
 Node nodeAfter = add(newEntry, currentNode.getNextNode());
 currentNode.setNextNode(nodeAfter);
 } // end if
 return currentNode;
} // end add
```

私有方法 add 将 newEntry 添加到由 currentNode 开始的子链表中。接下来我们会跟踪并解释它的逻辑。



### 学习问题 7 使用刚刚给出的方法 add 重做学习问题 5。

17.14

跟踪在有序表开头位置的添加过程。假定 nameList 是如图 17-3a 所示的有序表的链表。调用 nameList.add("Ally") 将 Ally 添加到有序表中。这个添加操作发生在链表的开头位置。公有方法 add 将用 add("Ally", firstNode) 来调用私有方法 add。实参 firstNode 中的引用拷贝给形参 currentNode，故 currentNode 也指向链表的第一个结点，如图 17-3b 所示。

因为 Ally 将添加到链表的开头，故执行语句

```
currentNode = new Node("Ally", currentNode);
```

会为 Ally 创建一个新结点。将这个结点链接到原链表中，如图 17-3c 所示。注意到，firstNode 没有改变，虽然它是对应于形参 currentNode 的实参。

现在私有方法返回 currentNode 的值，公有方法 add 将该值赋给 firstNode。这样，完成了添加后的链表如图 17-3d 所示。

跟踪有序表内部的添加：递归调用。当添加不在原链表的开头时会发生什么呢？让我们跟踪将 Luke 添加到图 17-4a 所示的链表时的情况。公有方法 add 通过 add("Luke", firstNode) 来调用私有方法 add。与前一段一样，firstNode 中的引用拷贝给形参 currentNode，故 currentNode 也指向链表中的首结点，如图 17-4a 所示。

17.15

因为 Luke 位于 Bob 的后面，故再一次递归调用：

```
add("Luke", currentNode.getNextNode())
```

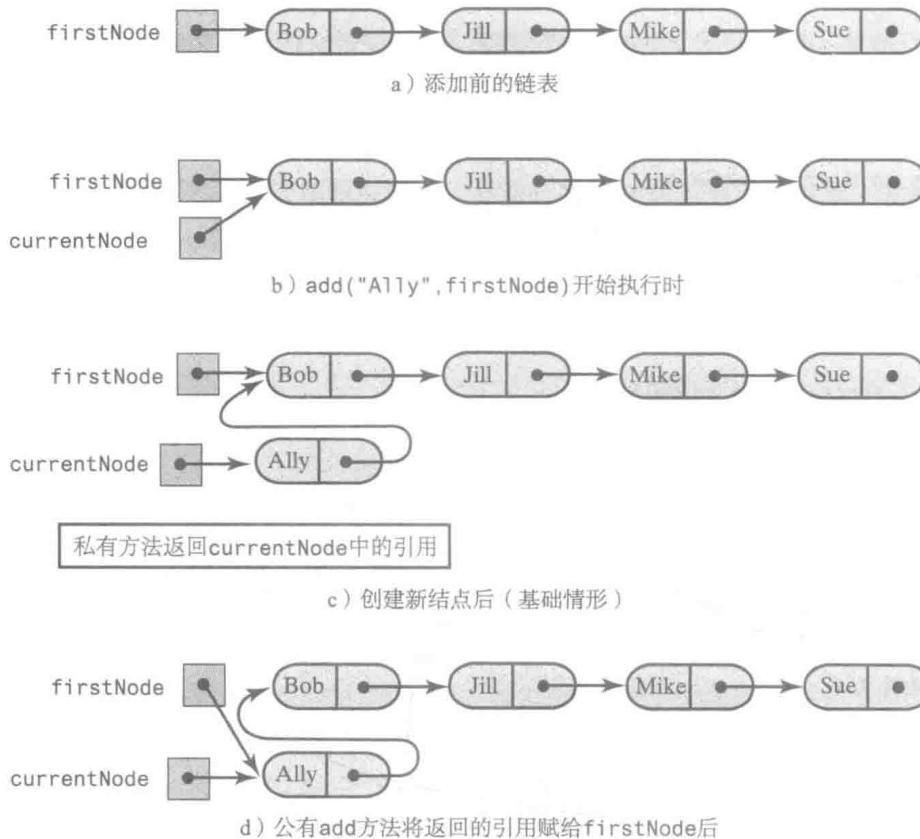


图 17-3 递归地在链表的开头添加结点

第二个参数是指向链表中第二个结点的引用，这个结点中含有 Jill。这个引用拷贝给形参 `currentNode`，如图 17-4b 所示。

Luke 位于 Jill 的后面，所以再次重复递归处理，`currentNode` 指向链表的第三个结点——Mike 所在的结点——如图 17-4c 所示。Luke 小于 Mike，故不再进行递归调用。已经到达基础情形。创建含有 Luke 且指向 Mike 所在结点的新结点，如图 17-4d 所示。

#### 17.16

跟踪从递归方法的返回。刚刚创建了一个新结点后，私有方法 `add` 返回指向它的引用，如图 17-4d 所示。调用 `add` 的语句现在恢复执行：

```
nodeAfter = add("Luke", currentNode.getNextNode());
```

所以，指向含有 Luke 的新结点的引用赋给 `nodeAfter`，如图 17-4e 所示。

此时，`currentNode` 指向 Jill 所在的结点，如图 17-4b 所示。下一条要执行的语句是

```
currentNode.setNextNode(nodeAfter);
```

故 Jill 所在结点的 `next` 数据域改为指向 Luke 所在的结点，如图 17-4f 所示。

现在私有方法 `add` 返回指向 Jill 所在结点的引用。如果我们继续跟踪，会令 Bob 所在结点指向 Jill 所在的结点，而 `firstNode` 指向 Bob 所在的结点，即使这些引用都已经就位。



**注：**向结点链表中进行递归添加，找到并记住插入点前的结点。位于插入点后面的链表的其余部分链接到新结点后，递归地将记住的结点链表接回链中。

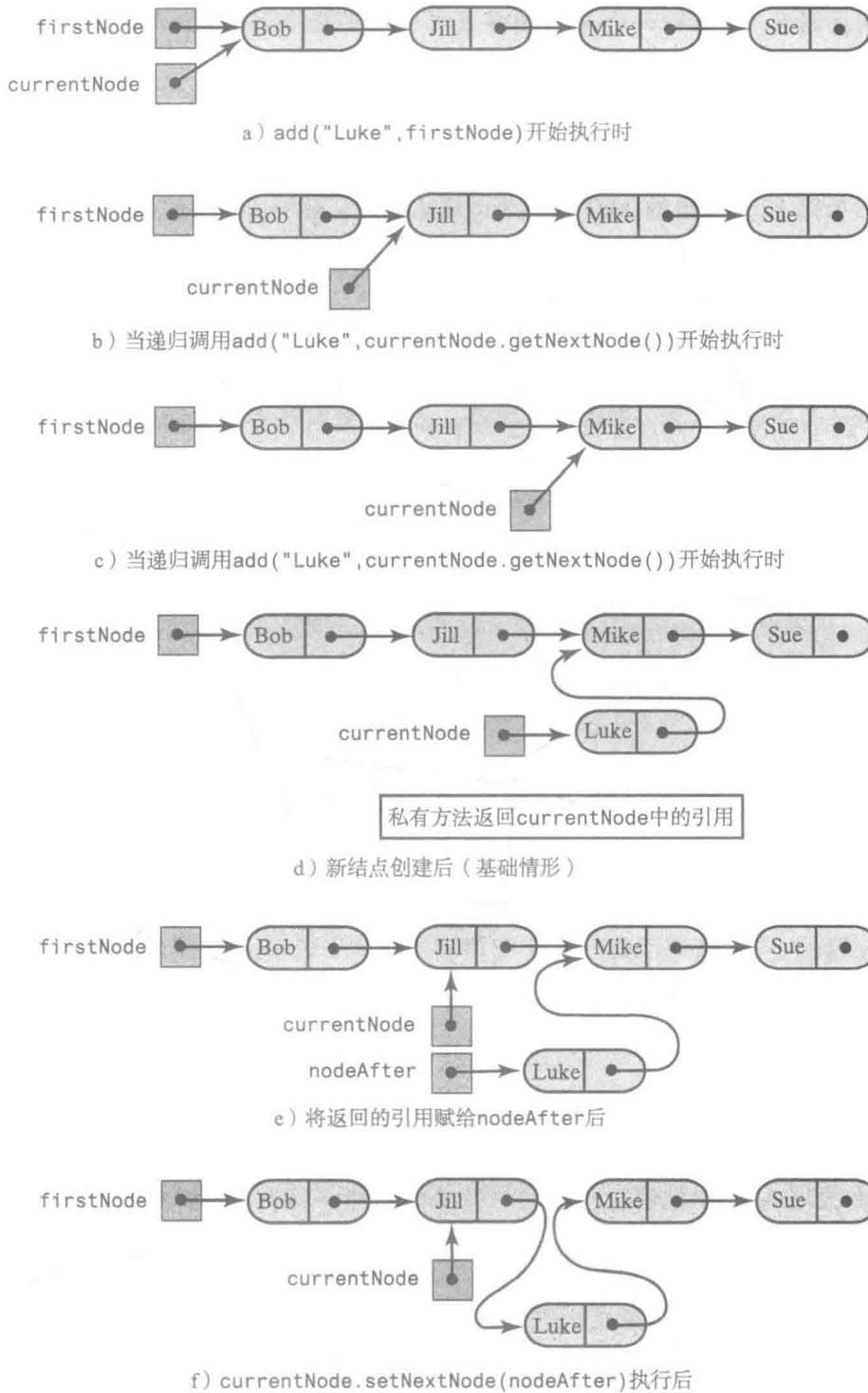


图 17-4 在链表的已有结点之间递归添加一个结点

本章结尾的项目 1 和项目 2，要求你完成有序表的迭代和递归实现。注意到，有序表的许多操作都与 ADT 线性表相同，所以这些实现与第 12 章看到的一样。

17.17

 注：因为 ADT 有序表与线性表有许多相同的操作，故它们有一部分实现是一样的。



**学习问题 8** 本章给出的 ADT 有序表的链式实现，没有维护尾引用。为什么尾引用对 ADT 线性表的链式实现比对有序表的链式实现更有意义？

## 链式实现的效率

17.18 如果分析第 12 章给出的 ADT 线性表的链式实现，会明白，`add` 方法的性能依赖于方法 `getNodeAt` 的效率。后者通过遍历结点链表找到插入点。它是  $O(n)$  操作。有序表的 `add` 方法自己来做遍历，找到添加位置。这个遍历也是  $O(n)$  的，使得有序表的添加也是  $O(n)$  操作。

图 17-5 概括了有序表操作的性能。这些结果的推导留作练习。当对这些实现进行比较时，你会发现，发生最差情形的情况可能是不同的。例如，添加到基于数组的有序表时，最差情形出现在有序表的开头，而链式实现时，它发生在有序表表尾。

| ADT 有序表操作                                                                                        | 数组     | 链式     |
|--------------------------------------------------------------------------------------------------|--------|--------|
| <code>add(newEntry)</code>                                                                       | $O(n)$ | $O(n)$ |
| <code>remove(anEntry)</code>                                                                     | $O(n)$ | $O(n)$ |
| <code>getPosition(anEntry)</code>                                                                | $O(n)$ | $O(n)$ |
| <code>getEntry(givenPosition)</code>                                                             | $O(1)$ | $O(n)$ |
| <code>contains(anEntry)</code>                                                                   | $O(n)$ | $O(n)$ |
| <code>remove(givenPosition)</code>                                                               | $O(n)$ | $O(n)$ |
| <code>display()</code>                                                                           | $O(n)$ | $O(n)$ |
| <code>clear()</code> , <code>getLength()</code> , <code>isEmpty()</code> , <code>isFull()</code> | $O(1)$ | $O(1)$ |

图 17-5 ADT 有序表两种实现方式下各操作的最差效率

## 使用 ADT 线性表的实现

17.19 正如在段 17.17 中所说明的，ADT 有序表的链式实现与 ADT 线性表的对应实现有许多重复的地方。我们能不能避免这些重复工作而重用线性表的实现部分呢？对这个问题的回答是肯定的，马上你就会看到。

你当然能够使用 ADT 线性表按字典序来创建并维护一个字符串表。然后，当实现 ADT 有序表时很自然地考虑使用 ADT 线性表。一般地，可以从两种方法中二选其一来实现。这里我们将线性表作为实现有序表的类的数据域。图 17-6 显示了这样的一个有序表的示例。回忆附录 C 中的段 C.1，这个方法称为组成，且表示两个类之间存在 has-a 关系。第 18 章考虑第二种方法，使用继承从线性表派生有序表。

17.20 我们的类 `SortedList` 将实现 `SortedListInterface` 接口。类的开头将线性表声明为数据域，且定义了默认的构造方法。假定第 12 章讨论的类 `LList` 实现了用于 ADT 线性表的接口 `ListInterface`。故类的开头如下所示：

```
public class SortedList<T extends Comparable<? super T>>
 implements SortedListInterface<T>
{
```

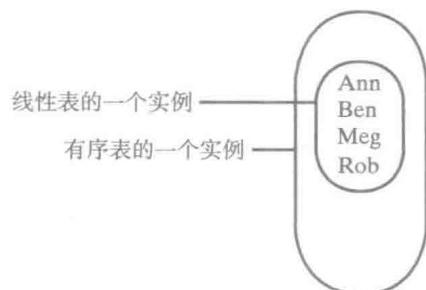


图 17-6 使用线性表保存项的一个有序表实例

```

private ListInterface<T> list;
public SortedList()
{
 list = new LList<>();
} // end default constructor
} // end SortedList

```

注意到，当声明数据域 list 时用到了泛型 T。

方法 add。ADT 有序表操作的实现都较短，因为大部分工作交由线性表完成。要将新项添加到有序表中，先使用方法 getPosition，这是有序表的一个操作。虽然目前我们还没有写出它，但假定它已经实现了。回忆一下，getPosition 找到已存在项在有序表中的位置，或是当有序表中没找到新项时，找到新项应该插入的位置。方法对返回的整数赋一个符号，以表示项是否存在于有序表中。当将一个项添加到允许含有重复项的有序表中时，项是否存在于有序表中是没有关系的。所以我们可以忽略 getPosition 返回的整数的符号。注意到，下列实现中使用类 Math 中的方法 abs 来丢弃符号。它还用到了 ADT 线性表的 add 操作。(本节中，对 ADT 线性表操作的调用都标记了出来。)

```

public void add(T newEntry)
{
 int newPosition = Math.abs(getPosition(newEntry));
 list.add(newPosition, newEntry);
} // end add

```



**学习问题 9** 使用刚给出的方法 add，重做学习问题 5。

**学习问题 10** SortedList 的客户能调用 ADT 线性表的操作 add(position, entry) 吗？解释之。

方法 remove。当从有序表中删除一个对象时，也用到 getPosition。但此时，我们必须知道给定项是否存在于有序表中。如果它不存在，则不能删除它。这种情形下，remove 将返回假。还要注意到，方法用到了 ADT 线性表的 remove 操作进行删除。所以，方法的实现如下：

```

public boolean remove(T anEntry)
{
 boolean result = false;
 int position = getPosition(anEntry);
 if (position > 0)
 {
 list.remove(position);
 result = true;
 } // end if
 return result;
} // end remove

```



**学习问题 11** 如果有序表含有 5 个重复对象，你使用前面的 remove 方法删除了其中的一个，从表中删除的是哪个：对象的第一次出现、对象的最后一次出现，还是对象的所有出现？

**学习问题 12** 前面这个用于有序表的 remove 方法，调用了线性表的 remove 方法，来删除给定位置的项。线性表的方法可能抛出一个异常，但我们并不捕获它。为什么在这里不需要捕获异常？

17.21

17.22

17.23 `getPosition` 的逻辑。`getPosition` 的实现比前两个方法的实现要难一些。要决定 `anEntry` 在有序表中的什么位置，必须从第一项开始将 `anEntry` 与有序表中已有的项进行比较。若 `anEntry` 已在有序表中，显然，在找到一个相等者之前一直进行项之间的比较。但是，如果 `anEntry` 不在有序表中，则我们想在项应位于有序表的位置处停止查找。使用类似于段 17.8 中描述的逻辑，从而利用对象的有序性。

例如，假定有序表中含有 4 个名字：Brenda、Carlos、Sarah 和 Tom。若想知道 Jamie 应该位于有序表的什么位置，发现作为字符串，

Jamie > Brenda

Jamie > Carlos

Jamie < Sarah

所以，Jamie 应该在 Carlos 的后面但在 Sarah 的前面——即有序表的位置 3，如图 17-7 所示。

要让 `anEntry` 与有序表中的项进行比较，先利用表操作 `getEntry` 返回有序表中给定位置的项。然后，使用表达式

`anEntry.compareTo(list.getEntry(position))`

进行比较。

17.24 `getPosition` 的实现。`getPosition` 的下列实现中，`while` 循环找到 `anEntry` 在有序表中的位置，`if` 语句查看 `anEntry` 是否在有序表中。

```
public int getPosition(T anEntry)
{
 int position = 1;
 int length = list.getLength();
 // Find position of anEntry
 while ((position <= length) &&
 (anEntry.compareTo(list.getEntry(position)) > 0))
 {
 position++;
 } // end while
 // See whether anEntry is in list
 if ((position > length) ||
 (anEntry.compareTo(list.getEntry(position)) != 0))
 {
 position = -position; // anEntry is not in list
 } // end if
 return position;
} // end getPosition
```

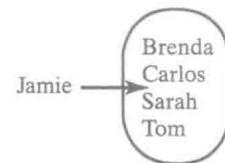


图 17-7 Jamie 在 Carlos 之后但在 Sarah 之前的有序表



学习问题 13 假定有序表 `nameList` 中含有作为字符串的 4 个名字，Brenda、Carlos、Sarah 和 Tom。跟踪 `getPosition` 的代码，看看当 `anEntry` 表示下列选项时，`getPosition` 返回什么

- a. Carlos      b. Alan      c. Wendy      d. Tom      e. Jamie

学习问题 14 因为根据 `getPosition` 返回的整数值的符号，能判定给定项是否在某个有序表中，所以可以使用 `getPosition` 来实现方法 `contains`。给出这个实现。

和 `toArray`——都与 ADT 线性表有相同的规范说明。每个方法都可以简单调用线性表中对应的方法。例如，`SortedList` 中方法 `getEntry` 的实现如下所示。

```
public T getEntry(int givenPosition)
{
 return list.getEntry(givenPosition);
} // end getEntry
```

 学习问题 15 可以通过调用学习问题 14 中提出的 `getPosition`，或是 ADT 线性表中的 `contains` 方法，来实现方法 `contains`。当要查找的项没有出现在有序表中时，这两个实现哪个执行得更快？为什么？

## 效率问题

也许除了 `getPosition` 中的某些微妙的逻辑之外，你可以快速写出前面的各个实现，即使有错误，错误也不会太多。使用已有的类建立另一个类的诱人之处是节省人力时间。但是，这样的实现会有效地利用计算机的时间效率吗？具体到这个实现，几个方法中都涉及 `getPosition`，所以这些方法的效率依赖于 `getPosition` 的效率。

`getPosition` 的效率。当我们检查段 17.24 中给出的 `getPosition` 时，注意到，线性表方法 `getLength` 是  $O(1)$  操作。所以，不需要在意它。而另一方面，循环语句中调用 `getEntry` 方法一次一个地检查线性表中的项，直到找到所需的项。所以 `getPosition` 的效率部分地依赖于 `getEntry` 的效率。但是，`getEntry` 的效率依赖于所使用的 ADT 线性表的实现。我们将考查线性表的两种实现，它们导致 `getPosition` 的效率截然不同。

第 12 章讨论了 ADT 线性表操作的效率。图 17-8 重新列出了线性表操作的最差情况效率，这是分析有序表时所必需的。如果使用数组表示线性表中的项，则 `getEntry` 永远是  $O(1)$  操作。所以 `getPosition` 中的循环最差情况下是  $O(n)$  的，当基于数组实现线性表时，`getPosition` 是  $O(n)$  的。

如果使用结点链表保存线性表中的项，则方法 `getEntry` 是  $O(n)$  的。因为 `getPosition` 的循环调用了 `getEntry`，所以我们得到，`getPosition` 在最差情况下是  $O(n^2)$  的。每次 `getEntry` 获取线性表中下一项时，都要从链表表头开始查找。这件事导致了 `getPosition` 的低效率。

| ADT 线性表操作                                              | 数组     | 链式     |
|--------------------------------------------------------|--------|--------|
| <code>getEntry(givenPosition)</code>                   | $O(1)$ | $O(n)$ |
| <code>add(newPosition, newEntry)</code>                | $O(n)$ | $O(n)$ |
| <code>remove(givenPosition)</code>                     | $O(n)$ | $O(n)$ |
| <code>contains(anEntry)</code>                         | $O(n)$ | $O(n)$ |
| <code>display()</code>                                 | $O(n)$ | $O(n)$ |
| <code>clear(), getLength(), isEmpty(), isFull()</code> | $O(1)$ | $O(1)$ |

图 17-8 ADT 线性表基于数组和链式实现下，几个操作的最差情况效率

`add` 的效率。段 17.21 中给出的有序表方法 `add` 的实现，含有下列语句：

```
int newPosition = Math.abs(getPosition(newEntry));
list.add(newPosition, newEntry);
```

对于 ADT 线性表基于数组的实现，`getPosition` 和线性表操作 `add` 都是  $O(n)$  的操作。

所以，有序表操作 `add` 在最差情况下是  $O(n)$  的。对于线性表的链式实现，`getPosition` 在最差情况下是  $O(n^2)$  的，而且是线性表操作 `add` 的主要部分，而后者仅为  $O(n)$ 。故有序表操作 `add` 在最差情况下是  $O(n^2)$  的。

17.28

图 17-9 概括了基于数组实现和链式实现 ADT 线性表时，有序表操作的效率。这些结果的证明留作练习。正如你看到的，本节给出的有序表的实现易于编写，但如果底层线性表使用了结点链表，那么效率并不高。第 18 章将介绍在不损失效率的前提下，如何重用 ADT 线性表来实现有序表。

### 学习问题 16 给出使用组成实现 `SortedList` 类时的优缺点。



| ADT有序表操作                                               | 线性表的实现 |          |
|--------------------------------------------------------|--------|----------|
|                                                        | 数组     | 链式       |
| <code>add(newEntry)</code>                             | $O(n)$ | $O(n^2)$ |
| <code>remove(anEntry)</code>                           | $O(n)$ | $O(n^2)$ |
| <code>getPosition(anEntry)</code>                      | $O(n)$ | $O(n^2)$ |
| <code>getEntry(givenPosition)</code>                   | $O(1)$ | $O(n)$   |
| <code>contains(anEntry)</code>                         | $O(n)$ | $O(n)$   |
| <code>remove(givenPosition)</code>                     | $O(n)$ | $O(n)$   |
| <code>display()</code>                                 | $O(n)$ | $O(n)$   |
| <code>clear(), getLength(), isEmpty(), isFull()</code> | $O(1)$ | $O(1)$   |

图 17-9 当使用 ADT 线性表实例实现时，ADT 有序表操作的最差情况效率



### 注：使用组成实现 ADT 有序表

当使用 ADT 线性表实例来表示 ADT 有序表的项时，必须使用线性表的操作来访问有序表的项，而不是直接访问它们。有序表的这种实现易于写程序，但当底层线性表使用结点链表保存项时，效率并不高。



**安全说明：**采用前面任一种实现的有序表实例，都容易改变其有序性。因为方法 `getEntry` 返回指向有序表中项的引用，客户可能——偶然地、不小心地或恶意地——修改了它的值，从而改变了表的有序性。一个防护办法是去掉公有方法 `getEntry`。另一个办法是，在有序表中仅放置不可变对象。要知道，如果向有序表中放置了可变对象，则恶意的客户可能会创建可变对象的一个子类，并修改其 `compareTo` 方法。

## 本章小结

- ADT 有序表按大小有序维护它的项。由它来决定将项放在哪里，而不是由客户决定。
- ADT 有序表可以添加、删除或查找一个项，给定的项作为实参。
- 有序表有几个操作，与 ADT 线性表对应的操作一样。但是，有序表不允许你按位置添加或替换项。
- 使用结点链表实现有序表时，效率还算合理。
- 使用 ADT 线性表作为数据域实现有序表时，程序易写。不过，根据 ADT 线性表的不同实现方式，其效率也不同。

## 练习

1. 假定 `nameList` 是名字的有序表。使用 ADT 线性表和 ADT 有序表的操作，创建这些名字的线性表，且不改变它们的次序。
2. 基于本章的规范说明，有序表能够含有重复项。给出含有唯一项有序表的规范说明。
3. 值的线性表的模（mode）是有最大频度的值。
  - a. 写算法，仅使用 ADT 有序表的方法，找到有序表的模。
  - b. 如果有序表采用数组实现，则算法的大  $O$  表示是什么？
  - c. 如果有序表采用链式实现，则算法的大  $O$  表示是什么？
4. 一个房间的活动时间表包含一个活动表（activity list）。每个活动有描述、起始时间和结束时间。可以将活动添加到表中，但它们必须与其他活动相容。如果两个活动的时间区间有重叠，则它们是不相容的。给出 ADT 活动表的规范说明。
5. 假定你与地质学家一起工作，他记录下过去 50 年间发生的地震。每个记录包括日期、地点、强度和持续时间。为这个数据集合设计并规范说明一个 ADT。
6. 解释如何使用 ADT 有序表来实现练习 4 和练习 5 中描述的 ADT。
7. 考虑基于数组实现的有序表。为实现方法 `add`，你必须将项添加到有序数组中，以使数组保持有序。
  - a. 描述实现步骤。
  - b. 你的逻辑是基于什么规范排序的？
  - c. 分析实现这个 `add` 的最差情况效率。
8. 图 17-5 将基于数组实现和基于链式实现的有序表操作的最差情况效率列表。推导这些大  $O$  表达式。
9. 图 17-9 将使用 ADT 线性表实例实现的有序表操作的最差情况效率列表。推导这些大  $O$  表达式。
10. 考虑有序表基于数组的实现。数组 `list` 是表示线性表项的数据域。如果给构造方法的参数是无序表项的数组，则构造方法必须将它们按有序放到 `list` 中。为此，它可能重复地使用有序表的 `add` 方法将项按合适的次序添加到有序表（即数组 `list`）中。或者它将项拷贝到 `list` 中，然后使用第 15 章和第 16 章的排序算法再排序它们。
  - a. 如果使用第一种方法，实际上使用的是什么排序方法？
  - b. 你会想用第二种方法吗？解释之。
11. 考虑使用 ADT 线性表的实例实现的有序表。具体来说，考虑方法 `contains`。`contains` 的一种实现是可以调用 `getPosition`（见段 17.24 结尾处的学习问题 14）。另一种实现是只调用 `list.contains`。比较这两种实现的效率。
12. 写有序表方法 `contains` 的链式实现。当它在链表中找到所需的项，或是越过项应该出现的位置后，查找应该结束。
13. 比较练习 12 中描述的方法 `contains`，与 `contains` 的线性表版本的效率。
14. 第 16 章段 16.2 中描述如何将两个有序数组归并到一个有序数组中。为 ADT 有序表添加一个归并两个有序表的操作。用下列三种方法实现归并：
  - a. 仅使用有序表的操作。
  - b. 假定基于数组实现。
  - c. 假定基于链式实现。

## 项目

1. 完成本章开头的 ADT 有序表的链式实现。使用迭代替代递归。
2. 重做项目 1，使用递归完成。
3. 使用数组表示 ADT 的项来实现 ADT 有序表。使用可变数组，以使有序表在需要时可以变大。
4. 使用 `Vector` 的实例表示 ADT 的项来实现 ADT 有序表。第 10 章项目 1 要求你创建 ADT 线性表的类似实现。

5. 练习 2 要求你给出含唯一项的 ADT 有序表的规范说明。实现一个这样的 ADT。

6. 在实现 ADT 的类内定义一个内部类，为 ADT 有序表添加一个迭代器。

7.  $x$  的多项式 (polynomial) 是一个含有  $x$  的整数幂的代数表达式，如下所示：

$$P(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$$

$a$  称为系数 (coefficient)。多项式的次数 (degree) 是  $n$ ，是出现在  $P(x)$  中的  $x$  的最大幂次。虽然在  $n$  次多项式中  $a_n$  不能是 0，但其他任何一个系数都可以是 0。

规范说明 ADT 多项式，包含操作，如 `getDegree`、`getCoefficient`、`setCoefficient`、`add` 和 `subtract`。使用有序表实现这个 ADT。有序表不应该含有任何为 0 的系数。

8. 练习 3 要求你创建一个查找有序表模的算法。现在在有序表的实现中添加一个方法，找到线性表的模。方法头应该是

```
public T getMode()
```

以下列三种方式实现这个方法：

- a. 仅使用有序表操作。
- b. 假定基于数组实现。
- c. 假定基于链式实现。

9. 可以使用替换代码 (substitution code) 对信息进行编码。这种机制下，密钥 (key) 将每个字符映射为另一个字符。根据密钥替换明文 (plain-text) 信息中的每个字符，得到编码信息，或称密文 (cipher text)。

假定，给你一些密文，但没有密钥。破解这种密码的一种方法是，计算密文中字符的频率，然后基于典型英文文本中各字符的频度来猜测这个映射。写一个程序，从文件中读入字符，使用有序表找到每个字符的频度。

10. 使用有序表保存优先队列中的项，实现 ADT 优先队列。

11. 第 1 章段 1.21 将集合定义为一个不允许有重复项的包。使用有序表保存项，实现 ADT 集合。包含分别在第 1 章练习 5、练习 6 和练习 7 中描述的并、交和差操作。

12. 在某些计算机网络中，信息不能作为数据的连续流发送。而是，将它们分成称为包 (packet) 的片段，一次发送一个包。包可能不能按它们发送的次序到达目的地。为使接收者能按正确的次序收集包，每个包中含有一个序列号。

例如，要发送信息 “Meet me at 6 o'clock”，每次发送 3 个字符，包应该如下所示：

```
1 Mee
2 t m
3 e a
4 t 6
5 o'c
6 l o c
7 k
```

不管包何时到达，接收者可以将包按它们的序列号排序，以确定信息。

给定一个含有按接收次序排列的数据包的文本文件，写一个应用，读入文件，并使用有序表提取信息。设计并创建辅助类，例如 `Packet` 和 `Message`。

13. 为用来报告本地之外新闻的在线网站设计一个新闻报料器。一旦记者有机会连接到互联网并上传故事，新闻事件就进入系统中。每个事件都有两个日期时间戳，一个表示事件的发生时间，另一个表示记录或上传故事的时间。

设计并实现一个事件对象，它有用于日期时间戳的两个数据域，以及详细描述事件的数据域。设计一个类，使用有序表按事件发生的次序维护事件。类应该能按顺序显示事件的时间轴，并提供重要的管理信息，如发生事件的时间和记录事件的时间之间的差。为用户提供一个选择，使得用户能按事件进入系统的顺序查看事件。

# 继承和多态

**先修章节：**序言、附录 C、第 6 章

本插曲延续附录 C 开始的关于继承的讨论，并介绍多态的概念。第 18 章当探讨 ADT 线性表的其他实现方式时会用到 Java 的这些内容。

## 继承的其他方面

到目前为止，本书已经用基本的有些直观的方式使用了继承。例如，在 Java 插曲 3 中写自己的异常类，它派生于 Java 类库中我们可用的标准异常类。虽然附录 C 详细讨论了继承机制，但它没有充分考虑继承的含义及是否合适。下面我们探讨这些内容。

### 何时使用继承

 **示例：**VectorStack 应该继承于 Vector 吗？在第 6 章实现 ADT 栈时，将栈的项保存在标准类 Vector 的实例中，它使用组成来定义类 VectorStack。程序清单 6-3 中给出的这个类的开头是这样的： J7.1

```
public final class VectorStack<T> implements StackInterface<T>
{
 private Vector<T> stack;
```

所以，VectorStack 的实例含有 Vector 的一个实例。

假定我们使用继承从 Vector 派生 VectorStack，如下：

```
public final class VectorStack<T> extends Vector<T>
 implements StackInterface<T>
{ . . . }
```

得到的类除了有 StackInterface 中的方法外，还有 Vector 的所有方法。但是，Vector 的这些方法能让客户在栈的任何位置添加或删除项，所以违背了 ADT 栈的前提。我们得到的不是栈，而是一个增强版的向量。但栈不是向量。因为在栈和向量之间不具有 is-a 关系，我们不应该使用继承来定义 VectorStack。

第 5 章段 5.23 中介绍的 Java 类库中的 `java.util.Stack` 类确实是从 `Vector` 派生的。所以这个类的实例并不是一个真正的栈。

### 安全说明：限制继承的使用

设计一个类的规范说明时，或者说明未来它用作基类（超类），或者将它声明为终极类阻止它用作基类。终极类易于定义并验证它的安全性。不是终极类的类，其中的非终极方法可能会被攻击者恶意重写。

宁愿用组成好过继承。



### 安全说明：了解超类可以影响子类的行为

子类不能对自己的行为有绝对的控制权。写了子类后，它的超类可能被修改，所以会影响子类的行为。例如，攻击者可能修改被子类继承但没被子类重写的超类方法的定义。即使子类重写了继承的所有方法——因此使继承的目的无效——对超类的修改也会影响子类的行为。这些修改可能包括增加了新的公有方法，或是修改了被已有的公有方法调用的私有方法。对超类的这些修改可能有意无意地破坏了子类中所做的假设，并导致不易察觉的安全漏洞。

## 保护访问

J7.2

你知道，使用像 `public` 或是 `private` 这样的访问修饰符可以控制对类的数据域和方法的访问。如附录 B 所示，当类写在一个包中，且你想让类仅用于本包中的其他类时，可以完全省去访问修饰符。访问控制还可以有其他的选择：可以将访问修饰符 `protected` 用于方法和数据域。

使用 `protected` 修饰的方法或数据域，仅在下列情况中可以按名访问：

- 它自己的类定义 C 内
- 从 C 派生的任意类内
- 与 C 在同一包中的任意类内

即如果一个方法在类 C 内被标记为 `protected` 的，则在从类 C 派生的类内的任意方法中都可以调用它。但是，对于不是从 C 派生的类，或不与 C 在同一个包中的类来说，保护方法等同于它是私有的。

应该继续将所有的数据域声明为私有的。如果想让子类访问超类中的数据域，则在超类内定义保护的访问方法或赋值方法。

注意到，包访问比保护访问更受限制，它能让程序员在定义类时有更多的控制。如果控制包目录，则可以控制谁被允许访问包。

图 JI7-1 说明了不同类型的访问修饰符。

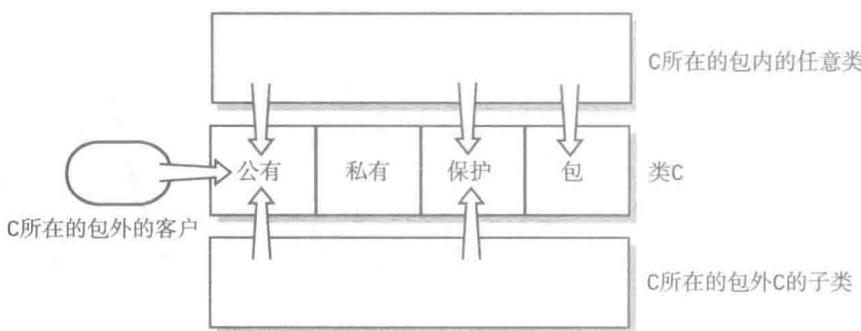


图 JI7-1 类 C 中公有、私有、保护和包访问的数据域及方法

## 抽象类和方法

J7.3

附录 C 的程序清单 C-1 中定义的类 `Student` 是其他类的超类，比如是程序清单 C-3 中给出的 `CollegeStudent` 类的超类。我们确实不需要创建 `Student` 类型的对象，虽然这样做肯定是合法的。但我们或许想阻止客户来创建 `Student` 类型的对象。为此，可以在类定

义的头部包含保留字 `abstract`，将类声明为抽象类（abstract class），如下所示：

```
public abstract class Student
{
```

注：抽象类是另一个类的超类。所以抽象类有时称为抽象超类（abstract superclass）。

当程序员定义一个抽象类时，他们常常声明一个或多个没有方法体的方法。这样做的目的，是要求每个子类按照子类中合适的方式来实现这些方法。例如，可能想让 `Student` 的每个子类实现方法 `display`。我们确实不能为尚未定义的未来的某个类写一个这样的方法，但我们可以要求一个。为此，在方法的头部包含保留字 `abstract`，将 `display` 声明为抽象方法（abstract method），如下所示：

```
public abstract void display();
```

注意到，方法头部的后面是一个分号，方法没有方法体。

注：抽象类内抽象方法的声明由方法的头部再加一个分号组成。头部必须包含保留字 `abstract`。抽象方法不能是私有的、静态的或终极的。

如果类内至少含有一个抽象方法，则 Java 要求类本身也要声明为抽象的。这是有意义的，否则你可能会创建一个未完成类的对象。在我们的示例中，对象会含有一个未实现的方法 `display`。J7.4

如果抽象类的子类没有实现所有的抽象方法又怎样呢？Java 将子类仍看作是抽象的，且阻止你创建这个类型的对象。例如，如果派生于 `Student` 的类 `CollegeStudent` 没有实现 `display`，则 `CollegeStudent` 也必须是抽象的。

注：至少含有一个抽象方法的类必须声明为抽象类。所以抽象方法仅能出现在抽象类内。

即使通过添加抽象方法 `display` 而让类 `Student` 是抽象的，也并不是它的所有方法都是抽象的。除了方法 `display` 之外的所有方法定义，都与原来的定义完全一样。它们都是完全定义好的，没有使用保留字 `abstract`。当在一个抽象类内实现一个方法有意义时，就应该这样做。用这种方式，可以在抽象类内尽可能多地包含细节，那些细节不必在子类内重复。

注：构造方法不能是抽象的

因为类不能重写其超类的构造方法，如果构造方法是抽象的，则它不能被实现。所以构造方法永远不是抽象的。

示例。让我们为类 `Student` 添加另一个方法，来调用抽象方法 `display`。先别抱怨调用了没有方法体的方法，要记着 `Student` 是一个抽象类。当最后从 `Student` 派生一个不是抽象类的类时，会实现 `display` 的。J7.5

我们提到的这个方法主要是当作示例，没做什么有用的事情。它仅仅是在显示一个对象

之前跳过指定的行数：

```
/** Displays the object after skipping numberofLines lines. */
public void displayAt(int numberofLines)
{
 for (int count = 0; count < numberofLines; count++)
 System.out.println();
 display();
} // end displayAt
```

方法 `displayAt` 调用抽象方法 `display`。这里抽象方法作为占位符留待未来的子类来定义。如果 `display` 不是抽象的，我们就必须给它一个方法体，而那个方法体真没什么用处，因为每个子类都要重写它。



**学习问题 1** 假定你将前一个方法 `displayAt` 的名字改为 `display`。得到的方法是重载还是重写方法 `display`？为什么？

## 接口与抽象类

J7.6



示例：对比抽象类与接口。序言中的段 P.17 定义了下列接口：

```
public interface Circular
{
 public void setRadius(double newRadius);
 public double getRadius();
} // end Circular
```

尽管没有提供表示半径的数据域，但这个接口声明了设置和获取方法，期待着实现这个接口的类会声明这个数据域的。事实上，段 P.17 中的类 `Circle` 实现了这个接口，声明了用于半径的数据域，定义了方法 `setRadius` 和 `getRadius`。它还定义了第三个方法 `getArea`。

现在不是去定义接口 `Circular`，而是来定义一个抽象类：

```
public abstract class CircularBase
{
 private double radius;
 public void setRadius(double newRadius)
 {
 radius = newRadius;
 } // end setRadius
 public double getRadius()
 {
 return radius;
 } // end getRadius
 public abstract double getArea();
} // end CircularBase
```

这个类声明了数据域 `radius`，这个类的后代类都要继承这个域。因为数据域 `radius` 是私有的，所以类 `CircularBase` 必须实现设置和获取方法，以便它的后代类都能访问这个数据域。如果 `CircularBase` 仅仅将 `setRadius` 和 `getRadius` 声明为抽象的——省略了它们的实现——则后代类将不能定义这些方法，因为后代类不能访问 `radius`。此外，如果 `CircularBase` 就只定义了这两个方法 `setRadius` 和 `getRadius`，那它没有必要是抽象的，但仍是一个有用的基类。不过，这个类还声明了抽象方法 `getArea`，后代类必须以自己的方式实现它。

下面的类派生于类 CircularBase。它实现了抽象方法 getArea，调用了继承的方法 getRadius 来访问继承的数据域 radius。Circle 不能按名使用数据域 radius。

```
public class Circle extends CircularBase
{
 public double getArea()
 {
 double circleRadius = getRadius();
 return Math.PI * circleRadius * circleRadius;
 } // end getArea
} // end Circle
```

在这个方法内，circleRadius 只是一个局部变量。

 **程序设计技巧：**如果想定义或声明类间共用的一个方法或一个私有数据域，使用抽象类。否则，使用接口。

## 多态

术语“多态”来源于希腊语，意思是“多种形式”。多态作为一个概念，实际上在英文中常见。例如，英语指令“做你最喜爱的运动”对不同的人意味着不同的事情。对某个人它意味着是打篮球。对另一个人它意味着是踢足球。在 Java 中，**多态** (polymorphism) 允许同一条程序指令在不同的上下文中意味着不同的事情。具体来说，一个方法名当作一条指令时，依据执行这个动作的对象类型，可能导致不同的动作。

起初，重载一个方法名可以看作多态。但是，该术语最新的用法是指，对于直接或间接重写的方法名，对象在运行时确定将使用方法的哪个动作。

### 注：多态

指令中的一个方法名可以根据调用该方法的对象的类型导致不同的动作。

 **示例。**例如，方法名 display 可以显示对象中的数据。但它显示的数据及显示多少，要依你用来调用该方法的对象的类型而定。我们为附录 C 的程序清单 C-1 中的 Student 类添加方法 display，假定方法和类都不是抽象的。故 display 在类 Student 内已经实现了。现在为类添加如段 J7.5 中所示的方法 displayAt。

如果涉及的只有一个类 Student，那么这些修改没什么令人激动的。但我们从类 Student 派生了 CollegeStudent，又从 CollegeStudent 派生了 UndergradStudent。类 UndergradStudent 从类 Student 继承了方法 displayAt。另外，UndergradStudent 重写了 Student 中定义的方法 display，提供了自己的实现。这样的话会如何呢？你或许已经困惑了。

好，来看看可怜的编译程序在遇到下列 Java 语句时的工作过程（我们忽略了指令的参数）：

```
UndergradStudent ug = new UndergradStudent(. . .);
ug.displayAt(2);
```

方法 displayAt 定义在类 Student 中，但它调用定义在类 UndergradStudent 中的 display 方法，如图 JI7-2 所示。甚至在类 UndergradStudent 定义之前，就能为类 Student 编译 displayAt 的代码。换句话说，编译好的这段代码可以使用 displayAt 被编译时甚至都还没有写的方法 display 的定义。这是如何做到的呢？

当编译 `displayAt` 的代码时, 对 `display` 的调用产生一条注解, 说“使用 `display` 的相应定义”。然后, 当调用 `ug.displayAt(2)` 时, 为 `displayAt` 编译的代码执行到这条注解, 并用与 `ug` 对应的 `display` 的版本来替换这条注解。因为这种情况下 `ug` 是 `UndergradStudent` 类型的, 所以 `display` 的版本是类 `UndergradStudent` 中定义的。

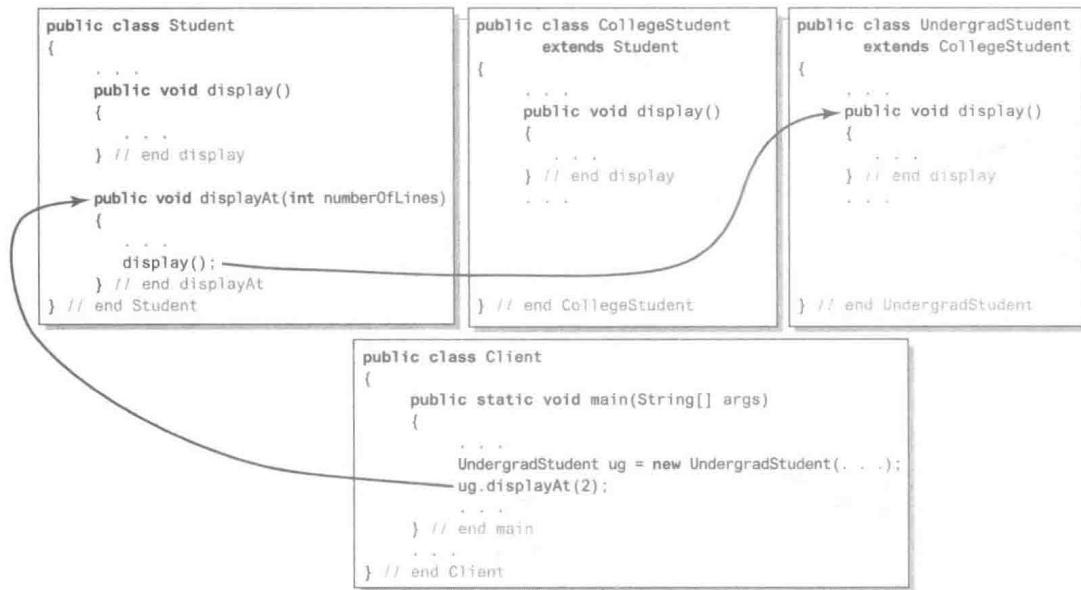


图 JI7-2 方法 `displayAt` 调用 `display` 的正确版本

### J7.9

决定使用哪个版本的定义, 依赖于接收对象在继承链中所处的位置, 而不是对象变量名的类型。例如, 考虑下列代码:

```
UndergradStudent ug = new UndergradStudent(...);
Student s = ug;
s.displayAt(2);
```

如附录 C 的段 C.21 所说明的, 将类 `UndergradStudent` 的一个对象赋值给 `Student` 类型的变量是完全合法的。这里, 变量 `s` 只是 `ug` 指向的对象的另一个名字。即 `s` 和 `ug` 都是别名。但对象仍记着它被创建为一个 `UndergradStudent`。这种情形下, `s.displayAt(2)` 最终会使用 `UndergradStudent` 中给出的 `display` 的定义, 而不是 `Student` 中给出的 `display` 的定义。

变量的静态类型 (static type) 是出现在声明中的类型。例如, 变量 `s` 的静态类型是 `Student`。静态类型是在代码编译时固定且确定下来的。运行时某一时刻变量指向的对象的类型称为动态类型 (dynamic type)。变量的动态类型随运行进程会改变。当执行前一段代码中的赋值语句 `s = ug` 时, `s` 的动态类型是 `UndergradStudent`。引用类型的变量称为多态变量 (polymorphic variable), 因为执行过程中, 它的动态类型可以不同于静态类型, 且可改变。

具体到我们的例子, Java 查看是哪个构造方法创建了对象, 从而确定要使用 `display` 的哪个定义。即 Java 使用变量 `s` 的动态类型, 来做出判断。



**注:** Java 使用对象的动态类型, 而不是它的名字, 来确定调用哪个方法。

调用稍后可能被重写的方法的这种处理方式称为动态绑定 (dynamic binding) 或后绑

定 (late binding)，因为在程序执行之前，方法调用的含义没有与方法调用的位置进行绑定。执行前面这段代码时，如果 Java 不使用动态绑定，就不会看到本科学生 (undergraduate student) 的数据。相反，你只能看到 Student 类提供的方法 display 所显示的内容。

### 注：动态绑定

动态绑定是，对同一个方法名，能让不同的对象使用不同的方法动作的过程。

Java 很棒，它能分清要使用方法的那个定义，即使类型转型也骗不过它。回忆一下，可以使用类型转型将一个值的类型转为其他的类型。即使使用类型转型将 ug 的类型改为类型 Student，前一段中 s.displayAt(2) 的含义也永远适用于 UndergradStudent 的对象，如下列语句所示：

```
UndergradStudent ug = new UndergradStudent(. . .);
Student s = (Student)ug;
s.displayAt(2);
```

尽管有类型转型，s.displayAt(2) 还是使用 UndergradStudent 中给出的 display 的定义，而不是 Student 中给出的 display 的定义。选择要调用的正确方法的决定因素是对象的动态类型，而不是它的名字。

为了明白动态绑定真的了不起，考虑下列代码：

```
UndergradStudent ug = new UndergradStudent(. . .);
Student s = ug;
s.displayAt(2);
GradStudent g = new GradStudent(. . .);
s = g;
s.displayAt(2);
```

标记出的两行是相同的，每一行都调用一个不同版本的 display。第一行显示一个 UndergradStudent，而第二行显示一个 GradStudent，如图 JI7-3 所示。对象能记住当用 new 运算符创建它时它所具有的方法定义。可以将对象赋给不同类（但需是祖先）类型的变量，不过，对于重写了方法的对象，选择哪个方法定义时是没有影响的。

我们继续深入探讨这个问题，来看看表象下面更加戏剧化的内容。注意到，类 UndergradStudent 和 GradStudent 的对象都从类 Student 继承了方法 displayAt，且都没有重写它。所以对于类 UndergradStudent 和 GradStudent 的对象，方法定义的内容甚至是相同的。被重写的是，在 displayAt 的定义中调用的方法 display。

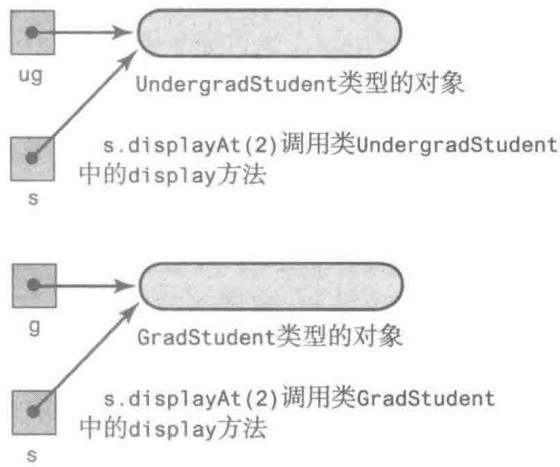


图 JI7-3 一个对象，而不是它的名字，决定它的行为

### 注：对象知道它们应该如何动作

当对象调用的方法，或者是一个重写的方法，或者是调用重写方法的方法，那个方法的动作由创建对象的构造方法所属的类来定义。选择什么动作不受命名对象的变量的

静态类型的影响。任意祖先类的变量都能指向后代类的对象，但对象永远会记得，对每个方法名使用哪个方法动作，因为 Java 使用动态绑定。

J7.11 类型检查和动态绑定。你必须要知道动态绑定如何与 Java 的类型检查互动。例如，如果 UndergradStudent 是 Student 类的一个子类，则可以将 UndergradStudent 类型的对象赋给 Student 类型的变量，如

```
Student s = new UndergradStudent();
```

但这还没有完。

虽然，可以将 UndergradStudent 类型的对象赋给 Student 类型的变量 s，但不能使用 s 来调用仅定义在 UndergradStudent 类内的方法。不过，如果在类 UndergradStudent 的定义中重写了一个方法，则会使用 UndergradStudent 内定义的这个方法版本。换句话说，变量决定使用哪个方法名，但对象决定使用方法名的哪个定义。如果想借由 Student 类型的变量 s 命名的对象，来使用首次定义在类 UndergradStudent 中的方法名，则必须使用类型转型。

J7.12  示例。例如，回忆一下，Student 不是抽象类，且实现了方法 display。还记得，UndergradStudent 是 Student 的子类。下列语句是合法的：

```
Student s = new UndergradStudent(. . .);
s.setName(new Name("Jamie", "Jones"));
s.display();
```

这里使用的是 UndergradStudent 类内给出的 display 的定义。记住，对象，而不是变量，决定将使用方法的那个定义。

另一方面，下列语句是不合法的：

```
s.setDegree("B.A."); // ILLEGAL
```

因为 setDegree 不是 Student 类内的方法名。记住，变量决定哪个方法名是可使用的。

变量 s 是 Student 类型的，但它指向 UndergradStudent 类型的一个对象。那个对象仍可以调用方法 setDegree，但编译程序不知道这一点。为了让调用合法，必须进行类型转型，如下这样：

```
UndergradStudent ug = (UndergradStudent)s;
ug.setDegree("B.A."); // LEGAL
```

你可能认为这只不过是一个愚蠢的练习，因为永远也不会将 UndergradStudent 类型的对象赋给 Student 类型的变量。不是这样的。这样直接的赋值可能不会经常进行，但不知不觉间的赋值却是经常发生的。回忆一下，我们可能将 UndergradStudent 类型的实参传给形参是 Student 类型的方法，那个形参就像是一个局部变量，且被赋予了对应实参的值。这种情形下，UndergradStudent 类型的对象（方法调用中的实参）就被赋给了 Student 类型的变量（方法定义中的形参）。

J7.13  示例。因为 Student 和 Name 都有各自版本的 toString 方法，所以可以如下显示这些类的对象：

```
Name joe = new Name("Joe", "Student");
Student s = new Student(joe, "5555");
System.out.println(s.toString());
```

感谢动态绑定，在调用 `System.out.println` 时甚至不需要写 `toString`。方法调用 `System.out.println(s)` 也有同样的效果，将得到同样的输出。下面来看看原因。

对象 `System.out` 有方法 `println`。方法 `println` 的一个定义中有唯一的 `Object` 类型的参数。这个定义等价于下列语句：

```
public void println(Object theObject)
{
 System.out.println(theObject.toString());
} // end println
```

花括号内调用的方法 `println`，是另一个方法，是方法 `println` 的重载定义，它带一个 `String` 类型而不是 `Object` 类型的形参。

`println` 的这些定义在 `Student` 类定义之前就已经存在。而带 `Student` 类型——所以也是 `Object` 类型——的对象 `s` 作为参数的调用

```
System.out.println(s);
```

使用的是 `Student` 的 `toString` 方法，而不是 `Object` 的 `toString` 方法。正是动态绑定来完成的这个工作。



示例：接口。你已经看到，如果类 B 是类 A 的子类，则可以写

J7.14

```
A item = new B();
```

变量 `item` 是多态的，因为它的动态类型可能不同于它的静态类型。多态还发生在使用接口时。例如，序言的段 P.20 中谈到了类 `Name` 和 `AnotherName`，它们两个都实现了接口 `NameInterface`。如果写

```
NameInterface myName = new Name("Jose", "Mendez");
```

则变量 `myName` 是多态的。它的数据类型是接口 `NameInterface`；`myName` 有 `NameInterface` 接口中的所有方法。例如，`myName.getFirst()` 将返回字符串 "Jose"。另外，如果现在写

```
myName = AnotherName("Maria", "Lopez");
```

则 `myName.getFirst()` 将返回字符串 "Maria"。所以使用继承或接口都可能导致多态变量。



学习问题 2 在类 `Student`、`CollegeStudent` 和 `UndergradStudent` 中都显式定义了不带参数的方法 `display`，这是重载的例子还是重写的例子？为什么？

学习问题 3 重载一个方法名是多态示例吗？

学习问题 4 下列代码中，`displayAt` 的两次调用会得到相同的输出吗？

```
Student s = new UndergradStudent(. . .);
s.displayAt(2);
s = new GradStudent(. . .);
s.displayAt(2);
```

## 继承和线性表

**先修章节：**附录 C、第 10 章、第 11 章、第 12 章、第 17 章、Java 插曲 7

### 目标

学习完本章后，应该能够

- 设计含有保护方法的类，让它适合用作基类
- 设计并使用一个抽象基类
- 使用继承高效实现有序表

第 17 章介绍了 ADT 有序表，它的项始终保持有序。与许多其他的 ADT 一样，可以使用数组或结点链表实现有序表。这样实现的优点在于它的时间效率。但是，这会与实现 ADT 线性表时有重复的部分，因为 ADT 有序表和线性表有几个共同的操作。

为了避免这些重复的工作，第 17 章使用 ADT 线性表实例来保存有序表的项。这个线性表是实现有序表类的数据域。这样实现起来很快，因为线性表的实现完成了大部分的工作。但因为有序表操作使用线性表的方式与客户使用线性表的方式是一样的，故当链式实现 ADT 线性表时，这些操作的效率并不高。

但如果不用第 17 章那样的组成，而是使用继承会如何呢？本章探讨从线性表派生有序表。这样做时，我们会发现一个子类（派生类）如果能访问其超类（基类）底层的数据结构，则它可以更有效率。如果超类中含有一些方法，能让未来的子类检查或修改其数据域，这是可行的。类的设计者应该为类的未来使用及当前的需求做出规划。

## 使用继承实现有序表

18.1 回忆我们在第 17 章段 17.20 的开头介绍的类 `SortedList` 的实现。`SortedList` 将另一个类的一个实例作为数据域，具体来说是 `LList`。`SortedList` 和 `LList` 具有 has-a 关系。`SortedList` 的几个方法（即 `remove`（通过位置）、`getEntry`、`contains`、`clear`、`getLength`、`isEmpty` 和 `toArray`）的行为类似于 `LList` 的方法。如果 `SortedList` 从 `LList` 继承了这些方法，则我们不必再实现它们，正如我们在第 17 章中所做的。所以可以修改 `SortedList` 如下：

```
public class SortedList<T extends Comparable<? super T>>
 extends LList<T> implements SortedListInterface<T>
{
 public void add(T newEntry)
 {
 int newPosition = Math.abs(getPosition(newEntry));
 super.add(newPosition, newEntry);
 } // end add

 <此处是remove(anEntry)和getPosition(anEntry)的实现代码>
 .
} // end SortedList
```

Java 插曲 5 中介绍过的表示法 `T extends Comparable<? super T>` 定义了泛型 `T`。

表示的类必须实现了接口 Comparable。符号 ? super T 代表 T 的任何超类，使得有序表所含有的对象的类型有一定的灵活度。

可以看到 SortedList 派生于 LList。还注意到，我们省略了段 17.20 中出现的数据域 list 和默认的构造方法。为修改段 17.21 中给出的 add 方法，我们只简单地将 list 替换为 super。即用下列语句

```
super.add(newPosition, newEntry);
```

来调用 ADT 线性表的 add 操作，以替代

```
list.add(newPosition, newEntry);
```

为一致起见，SortedList 的 add 方法重写了 LList 中将项添加到线性表表尾的 add 方法。

我们可对 remove 和 getPosition 方法进行类似的修改。有序表的其他方法都继承于 LList，所以它们不显式出现在 SortedList 中。

 **学习问题 1** 虽然 SortedList 继承了 LList 的 contains 方法，但方法没达到应有的效率。为什么？说明如何重写一个更有效率的 contains。

**陷阱。**这个实现中有一个陷阱，这是由使用继承导致的。虽然 SortedList 从 LList 方便地继承了像 isEmpty 这样的方法，但它也继承了客户可能会用来破坏有序表次序的两个方法。这两个方法出现在 ListInterface 中，如下所示：

```
/** Adds newEntry to the list at position newPosition. */
public void add(int newPosition, T newEntry);
/** Replaces the entry at givenPosition with newEntry. */
public T replace(int givenPosition, T newEntry);
```

例如，如果客户写如下的语句

```
SortedList<String> sList = new SortedList<>();
```

则它能使用 sList 来调用声明在 SortedListInterface 或 ListInterface 中的任何方法，包括前面的方法 add 和 replace。所以客户可以通过添加一个违反次序的项或是替换一个项，而破坏了有序表中项间的次序。

**避免陷阱的可能方法。**要避免这个陷阱我们能做什么呢？这里有 3 种可能：

- 在客户的有序表声明中使用 SortedListInterface。例如，如果客户含有语句

```
SortedListInterface<String> sList = new SortedList<>();
```

则使用 sList 仅能调用 SortedListInterface 中声明的方法。注意到，线性表操作 add 和 replace 没有出现在 SortedListInterface 中。虽然以这种方式使用 SortedListInterface 可能是一个良好的编程实用方法，但仅此而已。客户仅需要忽略这个实用方法，而将 sList 的数据类型定义为 SortedList，就能拥有 ADT 线性表中可以使用的所有操作。你已经见过了这种情形下客户是如何破坏有序表的。

- 在类 SortedList 中实现线性表的 add 和 replace 方法，给它们一个空的方法体。但是，调用方法的客户可能不知道方法没做任何事情。
- 在类 SortedList 中实现线性表的 add 和 replace 方法，当它们被调用时抛出一个异常。例如，add 方法可以像下面这样：

18.2

18.3

```

public void add(int newPosition, T newEntry)
{
 throw new UnsupportedOperationException("Illegal attempt to add " +
 "at a specified position within a sorted list.");
} // end add

```

这个版本的 `add` 方法也重写了 `LList` 实现的版本。如果客户调用这个方法，则会发生异常。这种方式是一种常用的实用方法，而且是我们喜欢的。



**注：**如果 `SortedList` 重写了线性表的方法 `add`，则类的实现仍能调用这个方法，就像前面那样处理即可。调用中使用 `super` 表示我们正在调用的是方法的线性表版本，而不是 `SortedList` 中重写的版本。



**学习问题 2** 刚刚给出的第 2 种可能方式进一步演变，可以在 `SortedList` 中实现 ADT 线性表的两个 `add` 方法，让每个方法都调用 `SortedListInterface` 中规范说明的 `add` 方法。以这种方式，新项将添加在有序表的正确位置。为什么这不是一个好主意？



**学习问题 3** `SortedList` 继承于其基类 `LList` 的方法 `toArray` 对有序表是不合适的。

a. 为什么这样说？

b. 写出 `SortedList` 中的方法 `toArray`，让它重写 `LList` 中的 `toArray` 方法。



**程序设计技巧：**如果你的类继承了不合适的方法，则可以重写它们，让它们在被调用时抛出一个异常。这种情况下，检查你的设计，并考虑继承是否是正确的选择。继承的好处是否超过了重写不合适的方法带来的不便，或者组成能提供更简捷的设计？

#### 18.4

**效率。**这里给出的 `SortedList` 的实现与第 17 章给出的使用组成的版本有同样的效率——或者具体来说是低效。如果当初设想着将 `LList` 设计为能被继承，那么 `SortedList` 就可以访问 `LList` 的底层数据结构，并提供更快的操作。为此，我们在下一节修改 `LList` 类。



**注：**派生于类 `LList` 的有序表的实现，与第 17 章给出的使用组成的实现同样低效。



**学习问题 4** 给出使用本节介绍的继承方式实现类 `SortedList` 的优缺点，每个方面至少列出一点。

## 设计一个基类

#### 18.5

现在来看看我们在第 12 章开发的作为 ADT 线性表的链式实现的类 `LList`。回忆一下，那个类将线性表的每个项放到自己的结点内。这些结点都链接起来，故第一项的结点指向第二项的结点，以此类推。类的数据域 `firstNode` 指向首结点，另一个数据域 `numberOfEntries` 记录线性表中的项数。

与大多数类一样，`LList` 有私有的数据域。客户不能直接通过名字来访问这些域。类的设计者必须决定是否为客户提供公有方法来间接访问这些数据域。具体到 `LList` 中，公有方法 `getLength` 能让客户得到线性表的长度。但是客户不能直接修改线性表的长度。只有其他的成员方法，例如 `add` 和 `remove`，才能改变其长度。另外，`LList` 没有提供对 `firstNode`

域的公有访问方法或赋值方法，从而不允许客户访问这个域。这样的设计是合适的，因为 `firstNode` 是实现细节，应该对客户隐藏。

这里所讨论的类的相关内容，反映在了程序清单 18-1 中列出的 `LList` 的摘要中。每个结点由一个私有类 `Node` 表示，它定义在 `LList` 中，并对客户隐藏。方法 `getNodeAt` 返回一个指向给定位置结点的引用，通过这个引用，方便了其他成员方法的实现。我们不想让客户访问这个结点，因为它是线性表的底层表示的一部分，所以我们让方法是私有的。

18.6

### 程序清单 18-1 类 `LList` 的相关内容

```

1 public class LList<T> implements ListInterface<T>
2 {
3 private Node firstNode; // Reference to first node
4 private int numberofEntries;
5
6 public LList()
7 {
8 initializeDataFields();
9 } // end default constructor
10
11 public void clear()
12 {
13 initializeDataFields();
14 } // end clear
15
16 <Implements of the public methods add, remove, replace, getEntry, contains,
17 getLength, isEmpty, and toArray go here.>
18
19 // Initializes the class's data fields to indicate an empty list.
20 private void initializeDataFields()
21 {
22 firstNode = null;
23 numberofEntries = 0;
24 } // end initializeDataFields
25
26 // Returns a reference to the node at a given position.
27 private Node getNodeAt(int givenPosition)
28 {
29
30 . . .
31
32 } // end getNodeAt
33
34 private class Node
35 {
36 private T data;
37 private Node next;
38
39 . . .
40
41 } // end Node
42 } // end LList

```

到目前为止，这些知识对你都是老调重弹。现在假定，我们想让 `LList` 用作你将开发的其他类的基类。在本章的前一节你已经看到，`LList` 的子类，正如 `LList` 的客户一样，不能按名字访问 `LList` 中声明为私有的任何成员。即子类不能访问数据域 `firstNode`、方法 `getNodeAt` 或类 `Node`，如图 18-1 所示。如果你想扩展 `LList` 的能力以使其更高效，则子类必须能访问类的这些成员——换句话说，这些底层数据结构。

18.7



图 18-1 类 LList 的派生类不能访问或修改 LList 内的私有成员

我们可以修改 LList，让它为子类提供可控的访问能力，允许访问那些向客户隐藏的项，使它更适合作为基类。首先，回忆保护访问，这在 Java 插曲 7 的段 J7.2 中讨论过。

### 注：保护访问

仅在自己的类 C 定义内、在 C 的任何子类内或与 C 同一包的类内，可以按名访问保护方法或数据域。

我们的目标是为子类提供保护的——但受限的——访问底层结点链表的能力。子类应该能高效地遍历或修改链表。但是，对链表的修改方式必须有助于维护其完整性。

18.8

为了允许子类能够按名访问不让客户访问的数据域，可以将 `firstNode` 和 `numberOfEntries` 声明为保护的。但更一般的做法是，让它们是私有的，并仅对我们所希望的访问提供保护方法。子类必须能访问头引用 `firstNode`，所以我们提供一个保护的获取方法来做这件事。因为 `getLength` 是公有的，子类能得到 `numberOfEntries` 的值。

子类可能需要修改 `firstNode` 和 `numberOfEntries`，所以我们提供保护方法来完成这件事。不过，我们想要一个高效的子类的同时，也想保持数据结构的完整性。所以不允许子类直接修改这些数据域，而是可以提供保护方法，按照符合我们要求的方式来修改结点链表。例如，保护方法可以添加或删除结点，并更新在此过程中链表的长度。为阻止子类重写这些保护方法，我们将它们声明为终极的。不提供能直接修改 `numberOfEntries` 域或者结点的链接部分的赋值方法。所以，子类可以高效地修改链表，但可以保证结点仍能正确链接，且链表的长度是准确的。

18.9

基于以上讨论，我们将 LList 修改如下。

1) 定义保护方法 `getFirstNode`，能让子类访问头引用 `firstNode`：

```

protected final Node getFirstNode()
{
 return firstNode;
} // end getFirstNode

```

2) 定义保护方法来添加及删除结点，必要时修改 `firstNode` 和 `numberOfEntries`。确保这些方法不能被重写，所以让它们成为终极方法是至关重要的，目的是保证底层数据结构的完整性，从而也保证了线性表的完整性。

```
/** Adds a node to the beginning of a chain. */
protected final void addFirstNode(Node theNode)

/** Adds a node to a chain after a given node. */
protected final void addAfterNode(Node nodeBefore, Node theNode)

/** Removes a chain's first node. */
protected final T removeFirstNode()

/** Removes the node after a given one. */
protected final T removeAfterNode(Node nodeBefore)
```

这些方法的实现用到了我们在第 12 章介绍的技术。例如，`addFirstNode` 的定义如下，假定 `theNode` 不是 `null`，且内部类 `Node` 有设置和获取方法：

```
protected final void addFirstNode(Node theNode)
{
 // Assertion: theNode != null
 theNode.setNextNode(firstNode);
 firstNode = theNode;
 numberOfEntries++;
} // end addFirstNode
```

3) `LList` 的公有方法和它的子类都能调用前面这些方法，所以减少了出错的可能。例如，可以修改 `LList` 的 `remove` 方法，如下所示：

```
public T remove(int givenPosition)
{
 T result = null;
 if ((givenPosition >= 1) && (givenPosition <= getLength()))
 {
 // Assertion: The list is not empty
 if (givenPosition == 1) // Case 1: Remove first entry
 result = removeFirstNode();
 else // Case 2: givenPosition > 1
 {
 Node nodeBefore = getNodeAt(givenPosition - 1);
 result = removeAfterNode(nodeBefore);
 } // end if
 return result; // Return removed entry
 }
 else
 throw new IndexOutOfBoundsException(
 "Illegal position given to remove operation.");
} // end remove
```

4) 接下来，让 `getNode` 是保护的及终极的，而不再是私有的。客户仍不能使用这个方法，但在类及任何子类的实现中可以使用。但是不能重写它进而来修改它。

5) 让类 `Node` 也是保护的及终极的，而不再是私有的。`Node` 仍对客户隐藏，但可让 `LList` 的任何子类使用。可以将 `Node` 的数据域 `data` 和 `next` 声明为保护的而不是私有的，但正如我们对 `LList` 所做的一样，将它们声明为私有的同时，提供保护的访问方法。我们还对结点的数据部分提供保护的设置方法。为确保链表的完整性，不允许子类来修改结点的链接部分，所以将这个设置方法声明为私有的。故 `Node` 有下面 4 个方法：

```
protected final T getData()
protected final void setData(T newData)
protected final Node getNextNode()
private final void setNextNode(Node nextNode)
```

最后，将 `Node` 的第一个构造方法声明为保护的，但将第二个构造方法声明为私有的，因为它设置了结点的链接部分。

对类 `LList` 做了这些修改后，得到一个新类，将其命名为 `LListRevised`。图 18-2 说明了这个类及派生类对它的访问。

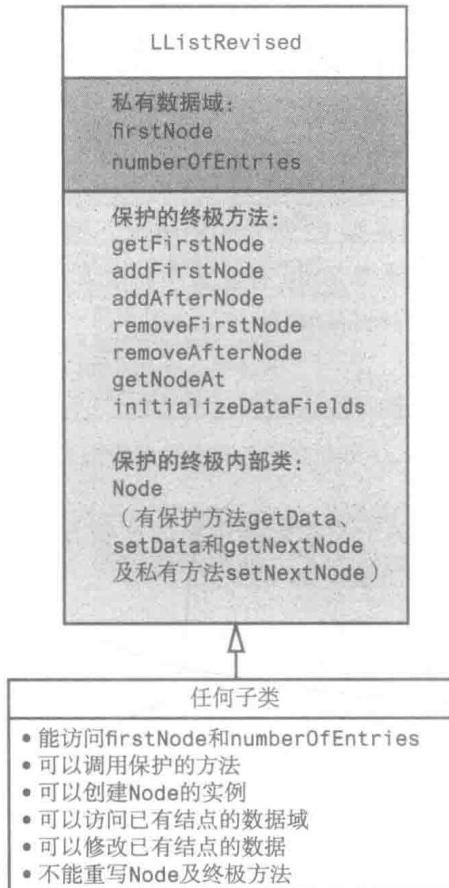


图 18-2 派生于类 `LListRevised` 的类能访问的

### 程序设计技巧：对未来的规划

当设计类时，应该规划当前的需求也要规划未来的使用。如果设计中没有公有的访问方法，则要提供保护的访问方法。确定是否想让未来的子类操作类的数据域。如果想这样做，则提供保护的方法，以便子类可以高效且安全地修改数据域。

### 学习问题 5 假定有类 `LListRevised` 的一个子类。

- 实现子类中的一个方法，将一个项添加到线性表的开头。
- 实现子类中的一个方法，将一个项添加到线性表中位项的右面。如果线性表含有  $n$  个项，则中位项是  $n/2$  位置的项，其中除法是整除。

## 创建抽象基类

18.10 可以将结点链表处理为抽象基类，从而简化前面的类 `LListRevised`。程序清单 18-2 概述了这样的一个类 `LindedChainBase`。注意到，这个类因有关键字 `abstract` 从而是抽象的。

象的。所以，虽然类的所有方法都已实现，但客户不能创建这个类的实例。

### 程序清单 18-2 抽象基类 LindedChainBase

```

1 public abstract class LinkedChainBase<T>
2 {
3 private Node firstNode; // Reference to first node
4 private int numberOfEntries;
5
6 public LinkedChainBase()
7 {
8 initializeDataFields();
9 } // end default constructor
10
11 < Implementations of the public methods clear, getLength, isEmpty, and toArray go here. >
12
13 < Implementations of the protected, final methods getFirstNode, addFirstNode,
14 addAfterNode, removeFirstNode, removeAfterNode, getNodeAt, and
15 initializeDataFields go here. >
16
17 protected final class Node
18 {
19 private T data; // Entry in list
20 private Node next; // Link to next node
21
22 protected Node(T dataPortion)
23 {
24 data = dataPortion;
25 next = null;
26 } // end constructor
27
28 private Node(T dataPortion, Node nextNode)
29 {
30 data = dataPortion;
31 next = nextNode;
32 } // end constructor
33 < Implementations of the protected methods getData, setData, and getNextNode go here. >
34
35 < Implementation of the private method setNextNode goes here. >
36
37 } // end Node
38 } // end LinkedChainBase

```



**安全说明：**为了保护 `LinkedChainBase` 的子类对象的任意线性表的完整性，我们不允许子类对底层的结点链表进行直接的添加或删除。另外，子类也不能直接修改 `numberOfEntries` 域。这样处理后，`LinkedChainBase` 的任何子类就不会因 `numberOfEntries` 的值与链表中当前的结点个数不一致而破坏了数据结构的完整性。要知道这样的破坏虽然可能无意，但后果很严重。

现在从 `LindedChainBase` 派生，并利用 `LListRevised` 中剩余的部分形成类 `LinkedChainList`，列在程序清单 18-3 中，如图 18-3 所示。

### 程序清单 18-3 派生于 `LindedChainBase` 的 `LinkedChainList`

```

1 public class LinkedChainList<T> extends LinkedChainBase<T>
2 implements ListInterface<T>
3 {
4 public LinkedChainList()

```

```

5 {
6 super(); // Initializes the linked chain
7 } // end default constructor
8
9 <Implementations of the public methods add, remove, replace, getEntry, and contains
10 go here.>
11 . . .
12 } // end LinkedChainList

```

`LinkedChainList` 提供了线性表方法，而不关心底层结点链表的细节。

基类 `LinkedChainBase` 也能用于其他的环境中。可用它来高效实现 ADT 有序表，接下来你会看到。



图 18-3 将链表操作与线性表操作分开



**注：**图 18-2 所示的 `LListRevised` 是一个复杂的类。它试图去描述一个线性表，同时为子类提供对底层结点链表的访问管理。像我们这样将这些工作分摊给多个类是一种常见的处理方法。这样，`LinkedChainList` 仅关注于线性表，而 `LinkedChainBase` 可以安全地管理链表。



学习问题 6 `LinkedChainList` 中的一些公有方法不能与 `LList` 中的定义完全相同。

- 所指的是哪些方法？
- 为什么 `LinkedChainList` 中的方法定义必须区别于 `LList` 中的定义？
- 在 `LList` 中的方法必须进行哪些修改，才能适合于 `LinkedChainList`？

## 有序表的高效实现

不是调用 ADT 线性表的操作来执行 ADT 有序表的操作，而是采用类似于我们在第 17 章段 17.7 开头处的链式实现，结果执行得会更快。定义在类 `LinkedChainBase` 中的保护方法能让我们对线性表底层数据结构的操作速度，比单纯依赖于 ADT 线性表的操作速度要快。所以我们想让类派生于 `LinkedChainBase`。由下列方法头开始：

```
public class LinkedChainSortedList<T extends Comparable<? super T>>
 extends LinkedChainBase<T>
 implements SortedListInterface<T>
```

与前面一样，我们将实现 `Comparable` 对象的有序表。

### 方法 add

新类中的 `add` 方法非常类似于段 17.10 中所给的用于 `LinkedSortedList` 类的方法。18.13但是，前面的添加操作的细节现在隐藏在 `LinkedChainBase` 的保护方法 `addFirstNode` 和 `addAfterNode` 中。所以我们修改的方法如下所示（对段 17.10 中 `add` 方法的修改已标出）：

```
public void add(T newEntry)
{
 Node theNode = new Node(newEntry);
 Node nodeBefore = getNodeBefore(newEntry);
 if (nodeBefore == null) // No need to call isEmpty
 addFirstNode(theNode);
 else
 addAfterNode(nodeBefore, theNode);
} // end add
```

每个保护方法前面的 `super` 是可选的，因为没有其他方法与它们同名。

第 17 章段 17.11 中的学习问题 4 提出的用于空线性表的简化也用在了这里。当线性表为空时，`getNodeBefore` 返回 `null`。所以可以省去在 `if` 语句中调用 `isEmpty`。

私有方法 `getNodeBefore`。我们仍然需要实现私有方法 `getNodeBefore`。这个实现类似于段 17.11 中所给的，但它使用的是 `getFirstNode()` 而不是 `firstNode`：18.14

```
private Node getNodeBefore(T anEntry)
{
 Node currentNode = getFirstNode();
 Node nodeBefore = null;
 while ((currentNode != null) &&
 (anEntry.compareTo(currentNode.getData()) > 0))
 {
 nodeBefore = currentNode;
 currentNode = currentNode.getNextNode();
 } // end while
 return nodeBefore;
} // end getNodeBefore
```

效率。方法 `add` 的这个版本，比段 18.1 和段 17.21 中所给的版本执行得要快。之前的18.15

版本仅使用了 ADT 线性表的操作，即类 `LList` 的公有方法。回忆一下，那些 `add` 方法先调用 `getPosition` 找到新项在线性表中的位置，然后它们调用线性表的 `add` 方法。段 17.24 中给出的 `getPosition` 方法遍历有序表，以找到新项的位置。执行这个遍历的  $O(n)$  次循环中将调用方法 `getEntry`。当链式实现 `getEntry` 时，它也遍历有序表，所以它是  $O(n)$  的。故 `getPosition` 是  $O(n^2)$  的。由此，段 18.1 和段 17.21 中的 `add` 方法都是  $O(n^2)$  的。

我们在段 18.3 中改进的 `add` 方法，通过最多一次遍历结点链表而将新结点添加在合适的位置。即使方法必须使用保护方法来改变结点链表，但它一旦找到合适的位置就可以添加新结点，而不需要重复地遍历链。所以它是  $O(n)$  操作。

18.16

**类的其他方法。**要实现 `remove` 和 `getPosition`，我们可以对其链式实现进行类似的修改。回忆一下，第 17 章将这些实现留作练习。我们还需要实现与 ADT 有序表共用的 ADT 线性表的其他操作，这些方法并不是从 `LinkedChainBase` 继承来的。这些方法是 `getEntry`、`contains` 和 `remove`（按位置）。最后，必须重写继承的方法 `toArray`，因为它分配一个不能转型为 `Comparable` 对象的对象数组。



注：如果你的基类为底层数据结构提供了保护的访问，则可以使用继承并能保持效率。



注：ADT 的实现可能不止一个。当选择给定应用的某种具体实现时，应该考虑与你的情形相关的所有因素。执行时间、内存使用及扩展性都是你应该考虑的因素。这些因素也应该是你实现一个 ADT 时要评估的。

## 本章小结

- 本章说明了使用组成及使用继承实现时的不同。基本思想与附录 C 中描述的一致的。使用组成时，类使用一个对象作为数据域。类的方法必须当作对象的客户，故它们仅能使用对象的公有方法。使用继承时，类继承了其基类的所有公有方法。它的实现及它的客户，都能使用这些公有方法。
- 基类可以提供保护方法，能让子类以客户不能的方式操作数据域。这种方式下，比起像客户那样不得不使用公有方法的方式，子类的方法更高效。
- 可以从有合适的保护方法的基类派生有序表，并仍有高效的实现。

## 程序设计技巧

- 如果你的类继承了不合适的方法，则可以重写它们，让它们在被调用时抛出一个异常。这种情况下，检查你的设计，并考虑继承是否是正确的选择。继承的好处是否超过了重写不合适的方法带来的不便，或者组成能提供更简捷的设计？
- 当设计类时，应该规划当前的需求和未来的使用。如果设计中没有公有的访问方法，则要提供保护的访问方法。确定是否想让未来的子类操作你的类的数据域。如果想这样做，则提供保护的方法，以便子类可以高效且安全地修改数据域。

## 练习

1. 为类 `LinkedChainSortedList` 实现方法 `contains`，如段 18.12 中所述。利用线性表的有序特性。

2. 为类 `LinkedChainSortedList` 写构造方法，如段 18.12 中所述，它有一个形参，是实现了 `ListInterface <T extends Comparable<? super T>>` 的类的实例。新的有序表应该含有线性表中的所有项，但要按序排列。
3. 为类 `LinkedChainSortedList` 写 `equals` 方法，如段 18.12 中所述，它重写了继承于 `Object` 类的 `equals` 方法。假定线性表中的对象都有 `equals` 的适当的实现，如果线性表中的每个项等于第二个线性表中对应的项，则你的新方法应该返回真。
4. 为段 18.10 中所述的类 `LinkedChainBase`，而不是为类 `LinkedChainSortedList`，重做练习 3。
5. 如果类 `LinkedChainBase` 有方法 `iterator`，如第 13 章段 13.8 所述，则为类 `LinkedChainSortedList` 定义一个迭代器时要做哪些工作？
6. 比较段 18.13 中给出的有序表方法

```
public void add(T newEntry)
```

与线性表方法

```
public void add(int newPosition, T newEntry)
```

的时间效率。

7. 第 17 章练习 5 要求你设计一个用于地震记录集合的 ADT。你能用 `LinkedChainBase` 做基类，使用继承来实现这个 ADT 吗？必须重写哪些方法？使用组成是不是更合适？
8. 这次用段 18.12 中描述的类 `LinkedChainSortedList` 替代 `LinkedChainBase`，重做练习 7。
9. 假定段 18.10 中所给的类 `LinkedChainBase` 实现了 Java 插曲 4 段 J4.13 中描述的接口 `java.util.ListIterator`。如果 `LinkedChainBase` 是 `LinkedChainSortedList` 的基类，则哪个迭代器方法适用于有序表？

## 项目

1. 完成段 18.12 开头的类 `LinkedSortedList` 的实现。
2. 从段 18.11 所述的类 `LinkedChainList` 派生类 `LinkedSortedList`。这种方法的缺点是什么？
3. 第 17 章的练习 4 要求你设计一个 ADT 活动表。说明你如何通过继承下列基类来实现这样一个类的？
  - a. `LinkedChainBase`。
  - b. `LinkedChainList`。
4. 第 17 章项目 8 要求你实现方法 `getMode`。展示你如何为段 18.12 所描述的类 `LinkedSortedList` 来实现这个方法的。
5. 使用继承，从类 `LinkedChainList` 派生第 13 章段 13.9 描述的类 `LinkedListWithIterator`。
6. 定义包的类，它实现第 1 章程序清单 1-1 所给的接口 `BagInterface`，且是程序清单 18-2 所给的 `LinkedChainBase` 的子类。
  - a. 设计用于链式实现的抽象基类 `LinkedSQD_Base`。指明每个域和方法是否为公有的、保护的或是私有的，并解释为什么。
  - b. 派生于你的基类，实现每个 ADT 栈、队列和双端队列的类。
  - c. 定义并使用基于数组实现的抽象基类 `ArraySQD_Base`，重做 a 和 b。
8. 18.1 节考虑了 3 种方法，以规避继承于 ADT 线性表的按位置添加操作及按位置替换操作这些有序表的陷阱。重写 `add` 和 `replace`，从而修改段 18.1 描述的类 `SortedList`，两个方法的行为修改如下：

```
/** Adds newEntry to this sorted list in its correct sorted order, ignoring the
 * value of newPosition. */
public void add(int newPosition, T newEntry);

/** Removes the entry at givenPosition from this sorted list and
 * then adds newEntry in its correct sorted order. */
public T replace(int givenPosition, T newEntry);
```

9. 应用程序分析器评估软件的时间或空间复杂度。实现一个应用程序分析器，它使用模拟来评估 ADT 有序表的各种实现的时间复杂度。特别地，考虑一个按下列方式使用整数有序表的应用程序：

- 65% 的有序表操作是将新项添加到表中
- 10% 的有序表操作是从表中删除项
- 15% 的有序表操作是获取给定位置的项或是获取给定项所在的位置
- 10% 的有序表操作是测试一个项是否在表中

要开始模拟，先创建一个有序表，含有 5000 个随机生成的 0 ~ 10 000 之间的整数。然后，在一个循环内，随机生成一个 0 ~ 10 000 的整数 `anEntry`，并根据先前的百分比随机挑选一个有序表操作。接下来，对 `anEntry` 执行这个操作。模拟 10 万到 100 万次操作，并记录下所用的总时间。使用第 17 章的 `LinkedSortedList`，然后再使用本章项目 1 中的 `LinkedSortedList` 重复模拟。ADT 有序表的哪种实现执行得最好？

10. 重做前一个项目，但使用下列混合操作：

- 15% 的有序表操作是将新项添加到表中
- 10% 的有序表操作是从表中删除项
- 55% 的有序表操作是获取给定位置的项或是获取给定项所在的位置
- 20% 的有序表操作是测试一个项是否在表中

## 查 找

**先修章节：**第 4 章、第 9 章、第 10 章、第 11 章、第 12 章、第 17 章

### 目标

学习完本章后，应该能够

- 使用顺序查找方法在数组中查找
- 使用二分查找方法在数组中查找
- 在链表中顺序查找
- 说明查找的时间效率

人们常会查找东西，日期、伙伴或是一只找不到的袜子。实际上，查找是计算机为我们做的最常见的工作。想想你在互联网上查找了多少次吧。本章研究两种简单的查找策略：顺序查找和二分查找。实现 ADT 线性表或 ADT 有序表的方法 `contains` 时，可以使用这些策略。当数据有序时，二分查找通常比顺序查找快得多，且二分查找只用于数组不能用于结点链表。但是排序数据通常比查找它需要更多的时间。这个事实会影响你在给定情形下对查找方法的选择。

### 问题

与图 19-1 中的人一样，你可能在桌面上找钢笔，在衣橱中找毛衣，或在名单中看看有没有你的名字。在有许多项的集合中查找一个具体的项——称为**目标** (target)——是一项常见的任务。19.1

让我们在那个表中找找你的名字。如果 `nameList` 是 ADT 线性表的实例，其中的项是名字，则我们可以使用线性表的操作 `contains` 进行查找。回忆一下，这个方法是布尔值的，如果给定的项存在于线性表中，则它返回真。

`contains` 的实现依赖于存储线性表项的方式。第 11 章和第 12 章中给出的 ADT 线性表的实现中，一个是将线性表项保存在数组中，另一个是保存在结点链表中。我们先来看看数组方式。



### 在无序数组中查找

图 19-1 查找每天都发生

第 11 章段 11.13 中提到，对线性表的顺序查找，是将要找的项——目标——与线性表中的第一项进行比较，与第二项进行比较，以此类推，直到找到所需的项或是找完了所有的项但没成功。在线性表基于数组的实现方式中，我们查找包含线性表项的数组。可以用迭代或递归方式实现这个查找。本节讨论这两种方法并看看它们的效率。19.2

虽然第 11 章中的线性表实现中没有使用——并忽略——含线性表项数组的第一个元素，不过本章的例子将查找整个数组。

## 无序数组中的迭代顺序查找

19.3 如段 11.13 中所示的 `contains` 的实现，使用循环来查找数组，并忽略其第一个元素。使用类似于方法 `contains` 的逻辑，下面的静态方法在保存泛型类型 T 的整个对象数组中查找具体的对象 `anEntry`。

```
public static <T> boolean inArray(T[] anArray, T anEntry)
{
 boolean found = false;
 int index = 0;
 while (!found && (index < anArray.length))
 {
 if (anEntry.equals(anArray[index]))
 found = true;
 index++;
 } // end while
 return found;
} // end inArray
```

一旦在数组中找到了第一个与所要找的项相等的项，立即退出循环。这种情况下，`found` 值为真。另一种情况，如果循环中检查了数组中的所有项，但没找到与 `anEntry` 相等的项，则 `found` 值为假。图 19-2 是这两种退出情况的示例。为简单起见，我们在图中使用的是整数。

查看9:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$8 \neq 9$ , 所以继续查找

查看5:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$8 \neq 5$ , 所以继续查找

查看8:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$8 = 8$ , 所以查找方法已经找到8

查看9:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$6 \neq 9$ , 所以继续查找

查看5:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$6 \neq 5$ , 所以继续查找

查看8:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$6 \neq 8$ , 所以继续查找

查看4:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$6 \neq 4$ , 所以继续查找

查看7:  

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$6 \neq 7$ , 所以继续查找

没有能比较的项了，所以查找结束。6不在数组中。

a) 查找8, 成功

b) 查找6, 失败

图 19-2 数组中的迭代顺序查找



学习问题 1 修改前一个方法 `inArray`，让其返回数组中第一个等于 `anEntry` 的项的下标。如果数组不含有这样的项，则返回 -1。

**学习问题 2** 写一个静态方法 `inList`, 仅使用 ADT 线性表中的操作, 实现线性表的迭代顺序查找。如果给定的项在所给的线性表中, 则方法应该返回真。

## 无序数组中的递归顺序查找

顺序查找数组从查找数组的第一项开始。如果那个项是要找的项, 则查找结束。否则查找数组的其余内容。因为这个新查找也是顺序的, 又因为数组的其余内容比原数组要小, 所以得到了问题求解的递归描述。对, 还差一点。我们需要一个基础情形。空数组可以是基础情形, 因为它永远不会包含要找的项。

对于数组 `a`, 查找从 `a[0]` 到 `a[n-1]` 之间的  $n$  个元素, 从查找第一个元素 `a[0]` 开始。如果它不是我们要找的, 则需要查找数组的其余部分, 即查找从 `a[1]` 到 `a[n-1]` 之间的数组元素。一般地, 查找 `a[first]` 到 `a[n-1]` 之间的数组元素。为更具一般性, 我们查找 `a[first]` 到 `a[last]` 之间的数组元素, 这里  $first \leq last$ 。

下列伪代码描述了递归算法的逻辑:

```
在 a[first] 到 a[last] 间查找 desiredItem 的算法
if (没有待查找的元素)
 return false
else if (desiredItem 等于 a[first])
 return true
else
 return 在 a[first + 1] 到 a[last] 间的查找结果
```

图 19-3 说明了数组上的递归查找。

实现这个算法的方法需要形参 `first` 和 `last`。要为客户省去提供这些形参的细节, 故让方法 `inArray` 有与段 19.3 中一样的方法头, 将算法实现为 `inArray` 调用的私有方法 `search`。

```
/** Searches an array for anEntry. */
public static <T> boolean inArray(T[] anArray, T anEntry)
{
 return search(anArray, 0, anArray.length - 1, anEntry);
} // end inArray
// Searches anArray[first] through anArray[last] for desiredItem.
// first >= 0 and < anArray.length.
// last >= 0 and < anArray.length.
private static <T> boolean search(T[] anArray, int first, int last,
 T desiredItem)
{
 boolean found;
 if (first > last)
 found = false; // No elements to search
 else if (desiredItem.equals(anArray[first]))
 found = true;
 else
 found = search(anArray, first + 1, last, desiredItem);
 return found;
} // end search
```

 **学习问题 3** 使用前面这个 `search` 方法, 在下列对象数组中查找对象 `o`, 列出所做的比较。假定对象没有找到。

o1 o2 o3 o4 o5

**学习问题 4** 在客户层实现递归方法 search，用来查找对象线性表。仅使用 ADT 线性表中的操作。如果给定的项在所给的线性表中，则方法应该返回真。

查看第一项，9：

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$8 \neq 9$ ，所以查找紧接其后的子数组。

查看第一项，5：

|   |   |   |   |
|---|---|---|---|
| 5 | 8 | 4 | 7 |
|---|---|---|---|

$8 \neq 5$ ，所以查找紧接其后的子数组。

查看第一项，8：

|   |   |   |
|---|---|---|
| 8 | 4 | 7 |
|---|---|---|

$8 = 8$ ，所以查找方法已经找到8。

查看第一项，9：

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$6 \neq 9$ ，所以查找紧接其后的子数组。

查看第一项，5：

|   |   |   |   |
|---|---|---|---|
| 5 | 8 | 4 | 7 |
|---|---|---|---|

$6 \neq 5$ ，所以查找紧接其后的子数组。

查看第一项，8：

|   |   |   |
|---|---|---|
| 8 | 4 | 7 |
|---|---|---|

$6 \neq 8$ ，所以查找紧接其后的子数组。

查看第一项，4：

|   |   |
|---|---|
| 4 | 7 |
|---|---|

$6 \neq 4$ ，所以查找紧接其后的子数组。

查看第一项，7：

|   |
|---|
| 7 |
|---|

$6 \neq 7$ ，所以查找空数组。

没有要比较的项了，所以查找结束。 $6$ 不在数组中。

a) 查找8，成功

b) 查找6，失败

图 19-3 数组中的递归顺序查找

## 顺序查找数组的效率

19.7

不管是用迭代方式还是递归方式实现顺序查找，比较次数都是一样的。最好情况下，在数组的第一个位置找到所要找的项。此时将只进行一次比较，所以查找是  $O(1)$  的。最差情况下，将查找整个数组。或者是在数组的最后找到所需的项，或者是完全没找到。这两种情况下，对含  $n$  项的数组都要进行  $n$  次比较。所以顺序查找最差情况是  $O(n)$  的。一般地，要查找数组中差不多一半的项。所以，平均情况是  $O(n/2)$ ，这也是  $O(n)$  的。



注：数组中的顺序查找的时间效率

最优情况： $O(1)$

最差情况： $O(n)$

平均情况： $O(n)$

## 在有序数组中查找

对无序数组的顺序查找很容易理解及实现。当数组含有较少的项时，查找的效率尚可实用。但是当数组中含有 many 项时，顺序查找可能是很费时的。例如，假定你在一罐硬币中查找你出生那年铸造的。顺序查找 10 枚硬币不是个问题。1000 枚硬币的查找可能就久一些。

了；100万枚硬币的查找是巨久的。需要一个更快的查找方法。幸运的是，更快的查找是可能的。

## 有序数组中的顺序查找

假定在开始查找硬币之前，有人将它们按时间重排了。如果你在图19-4中所示的有序硬币中，顺序查找2000年的硬币，在到达2000之前先查看了1995、1997和1998年的。或者，如果是要查找1999年的，在查看了前4枚硬币后并没发现它。还继续查找吗？如果硬币已升序排列，而你到达了2000年的，则在后面不会找到1999年的。如果硬币没有排序，则必须检查所有的硬币才能发现1999年的那枚没有出现。

19.8

注：如果数据有序，则顺序查找能有更高的效率。



图19-4 按铸造日期排序的硬币

如果数组按升序或降序排序，则可以使用前面的思想来修改顺序查找。修改后的查找可以比无序数组的顺序查找更快地告诉我们查找项不出现在数组中。这种情形下，后一种查找总要检查整个数组。而对于有序数组，修改后的顺序查找常进行少得多的比较就能做出相同的判定。本章结尾的练习2要求你实现有序数组上的顺序查找。

在花了力气对数组进行排序后，使用我们接下来要讨论的方法，对它进行查找可能更快。

## 有序数组中的二分查找

考虑 $1 \sim 100$ 万之间的一个数。当我要猜你这个数时，告诉我我的猜测是正确的、太大还是太小。在正确猜到之前最多进行多少次尝试？当读到本段末尾时你肯定能回答这个问题。

19.9

如果要在印刷的号码簿中找一个新朋友的电话号码，你会怎么做？一般地，你会打开到接近中间的一页，扫一眼这些项，马上就能明白你有没有翻到正确的一页。如果没有，则决定是看更前面的页——那些页在书的左“半”边——还是更后面的页——那些页在右“半”边。电话号码簿的什么特点能让你这样决定？名字的字典序。

如果你决定在左半部分查找，则可以忽略整个右半部分。实际上，可以撕掉右半部分并丢掉它，如图19-5所示。当你只在书的左半部分查找时，已经有效地减少了查找问题的大小。然后重复处理这半部分。最终，可以找到电话号码，或是它并不存在。这个方法——称为二分查找（binary search）——听起来是递归的。

现在稍做修改，就可以将这个思想用于查找升序有序的n个整数的数组。（在算法中仅做简单修改就可用于降序有序。）已知



19.10

图19-5 当数据有序时忽略一半的数据

$a[0] \leq a[1] \leq a[2] \leq \dots \leq a[n-1]$

因为数组是有序的，所以可以排除掉数组中不可能包含所查找数的那个部分的全部——就像是你排除电话号码簿的一半一样。

例如，如果我们要查找 David，我们知道  $a[5]$  等于 Jaime，那当然知道 David 在  $a[5]$  之前。而我们还知道 David 不会出现在数组的  $a[5]$  之后，因为数组是有序的。即

"David" <  $a[5] \leq a[6] \leq \dots \leq a[n-1]$

我们知道，不需要在  $a[5]$  之后的元素中进行查找。所以这些元素与  $a[5]$  一起都可以忽略掉。类似地，如果要查找的名字位于  $a[5]$  之后（例如，如果要查找 Mia），则可能忽略  $a[5]$  和它之前的所有元素。

用数组中间位置的下标，替换前一个例子中的下标 5，得到对数组进行二分查找算法的初稿：

```
在 a[0] 到 a[n-1] 间查找 desiredItem 的算法
mid = 0 到 n-1 间的近似中点
if (desiredItem 等于 a[mid])
 return true
else if (desiredItem < a[mid])
 返回 a[0] 到 a[mid-1] 间的查找结果
else if (desiredItem > a[mid])
 返回 a[mid+1] 到 a[n-1] 间的查找结果
```

注意到要

查找  $a[0]$  到  $a[n-1]$

则必须或者

查找  $a[0]$  到  $a[mid-1]$

或者

查找  $a[mid+1]$  到  $a[n-1]$

这两个对部分数组的查找都是我们正解决的问题的更小版本，所以能够通过递归调用算法本身来完成。

19.11

但当我们写前面伪代码的递归调用时，又会引起其他问题。每次的调用都查找数组的一个子范围。第一种情况是查找下标 0 到  $mid-1$  之间的元素。第二种情况是查找下标  $mid+1$  到  $n-1$  之间的元素。所以，我们需要两个额外的形参——`first` 和 `last`——来说明要查找的数组子范围的第一个和最后一个下标。即在  $a[first]$  到  $a[last]$  之间查找 `desireditem`。

使用这些形参，且让递归调用更像 Java 语言风格，则将伪代码表示如下：

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = first 到 last 间的近似中间点
if (desiredItem 等于 a[mid])
 return true
else if (desiredItem < a[mid])
 return binarySearch(a, first, mid - 1, desiredItem)
else if (desiredItem > a[mid])
 return binarySearch(a, mid + 1, last, desiredItem)
```

为查找整个数组，初始时 `first` 设置为 0，`last` 设置为  $n-1$ 。然后每次递归调用时

`first` 和 `last` 将使用另外的值。例如，先出现的那个递归调用将 `first` 设置为 0，`last` 设置为 `mid-1`。

当写任何递归算法时，总应该检查递归不是无穷的。我们来看看每次合理的算法调用是否能导向基础情形。考虑前一段伪代码中嵌套的 if 语句中的三种情形。第一种情形，在数组中发现了要查找的项，所以没有递归调用，处理结束。其他两种情形中，通过递归调用去查找数组中更小的一部分。如果要查找的项在数组中，则算法使用越来越小的部分，直到找到这个项。但如果项不在数组中将如何？一系列递归调用的结果会导向基础情形吗？不幸的是，不会，但这不难修改。

注意，每次递归调用中，或者 `first` 的值增大了，或者 `last` 的值减小了。如果它们相互交错，`first` 实际上变得大于 `last` 了，则数组中再没有元素需要检查。那种情形下，`desiredItem` 不在数组中。如果将这个测试添加到伪代码中，则可以改善算法，得到更完整的算法，如下所示。19.12

```
Algorithm binarySearch(a, first, last, desiredItem)
 mid = (first + last) / 2 // Approximate midpoint
 if (first > last)
 return false
 else if (desiredItem 等于 a[mid])
 return true
 else if (desiredItem < a[mid])
 return binarySearch(a, first, mid - 1, desiredItem)
 else // desiredItem > a[mid]
 return binarySearch(a, mid + 1, last, desiredItem)
```

图 19-6 是二分查找的示例。

 学习问题 5 当用前一个二分查找算法在图 19-6 所示的数组中查找 8 和 16 时，分别进行了多少次与数组项间的比较？

查看中间项，10：

|   |   |   |   |   |           |    |    |    |    |    |    |
|---|---|---|---|---|-----------|----|----|----|----|----|----|
| 2 | 4 | 5 | 7 | 8 | <b>10</b> | 12 | 15 | 18 | 21 | 24 | 26 |
| 0 | 1 | 2 | 3 | 4 | 5         | 6  | 7  | 8  | 9  | 10 | 11 |

8 < 10，所以查找数组的左半部分。

查看中间项，5：

|   |   |          |   |   |
|---|---|----------|---|---|
| 2 | 4 | <b>5</b> | 7 | 8 |
| 0 | 1 | 2        | 3 | 4 |

8 > 5，所以查找数组的右半部分。

查看中间项，7：

|   |   |
|---|---|
| 7 | 8 |
| 3 | 4 |

8 > 7，所以查找数组的右半部分。

查看中间项，8：

|   |
|---|
| 8 |
| 4 |

8 = 8，所以查找结束，8 在数组中。

a) 查找 8，成功

图 19-6 有序数组上的递归二分查找

查看中间项，10：

|   |   |   |   |   |           |    |    |    |    |    |    |
|---|---|---|---|---|-----------|----|----|----|----|----|----|
| 2 | 4 | 5 | 7 | 8 | <b>10</b> | 12 | 15 | 18 | 21 | 24 | 26 |
| 0 | 1 | 2 | 3 | 4 | 5         | 6  | 7  | 8  | 9  | 10 | 11 |

16 > 10，所以查找数组的右半部分。

查看中间项，18：

|    |    |           |    |    |    |
|----|----|-----------|----|----|----|
| 12 | 15 | <b>18</b> | 21 | 24 | 26 |
| 6  | 7  | 8         | 9  | 10 | 11 |

16 < 18，所以查找数组的左半部分。

查看中间项，12：

|           |    |
|-----------|----|
| <b>12</b> | 15 |
| 6         | 7  |

16 > 12，所以查找数组的右半部分。

查看中间项，15：

|           |
|-----------|
| <b>15</b> |
| 7         |

16 > 15，所以查找数组的右半部分。

下一步的子数组为空，所以查找结束。16不在数组中。

b) 查找16，失败

图 19-6（续）

19.13

虽然段 19.3 和段 19.6 中给出的顺序查找的实现用到了方法 `equals` 进行必要的比较，但二分查找需要的不只是相等测试。为进行必要的比较，我们需要方法 `compareTo`。因为所有的类从 `Object` 类继承了 `equals` 并可重写它，故所有的对象都有方法 `equals`。但有 `compareTo` 方法的对象必须属于实现了接口 `Comparable` 接口的类。这正是第 17 章段 17.1 中说明的有序表中对象的情况，也是有序数组中对象的情况。

方法 `binarySearch` 的实现如下。

```
private static <T extends Comparable<? super T>>
 boolean binarySearch(T[] anArray, int first, int last, T desiredItem)
{
 boolean found = false;
 int mid = first + (last - first) / 2;
 if (first > last)
 found = false;
 else if (desiredItem.equals(anArray[mid]))
 found = true;
 else if (desiredItem.compareTo(anArray[mid]) < 0)
 found = binarySearch(anArray, first, mid - 1, desiredItem);
 else
 found = binarySearch(anArray, mid + 1, last, desiredItem);
 return found;
} // end binarySearch
```

现在公有方法 `inArray` 的实现如下。

```
public static <T extends Comparable<? super T>> boolean inArray(T anEntry)
{
 return binarySearch(anArray, 0, anArray.length - 1, anEntry);
} // end inArray
```



注：计算中间点 mid 的 Java 语句是

```
int mid = first + (last - first) / 2;
```

而不是

```
int mid = (first + last) / 2;
```

如第 9 章段 9.18 结尾处的注释中所讨论的。



**程序设计技巧：**实现了 Comparable 接口的类必须定义 compareTo 方法。这样的类还应该定义一个 equals 方法，它重写了继承于 Object 的 equals 方法。compareTo 和 equals 方法应该使用相同的相等测试。前面的方法 binarySearch 中调用了 equals 方法，也调用了 compareTo 方法。如果数组中的对象没有相应的 equals 方法，则 binarySearch 不会正确执行。不过，注意，可以使用 compareTo 来替代 equals 进行相等测试。



**学习问题 6** 二分查找中，当目标是 a.2；b.8；c.15 时，数组 4 8 12 14 20 24 中的哪些项与目标进行过比较？

**学习问题 7** 修改前面的方法 inArray，让其返回数组中第一个等于 anEntry 的数组项的下标。如果数组不包含这样的项，则返回 -1。你还必须修改 binarySearch 方法。

**学习问题 8** 当数组降序有序（从最大降到最小）而不是我们讨论中假定的升序有序时，查找算法必须进行哪些修改？

## Java 类库：binarySearch 方法

包 java.util 中的 Arrays 类定义了静态方法 binarySearch 的几个不同版本，其详细说明如下。19.14

```
/** Searches an entire array for a given item.
 * @param array An array sorted in ascending order.
 * @param desiredItem The item to be found in the array.
 * @return Index of the array entry that equals desiredItem;
 * otherwise returns -belongsAt - 1, where belongsAt is
 * the index of the array element that should contain desiredItem. */
public static int binarySearch(type[] array, type desiredItem);
```

这里，出现的两个 type（类型）必须是相同的；类型可以是 Object 或是任意的基本类型 byte、char、double、float、int、long 或 short。

## 数组中二分查找的效率

二分查找算法在只检查一个元素后排除掉大约一半的数组元素。然后它再排除掉数组中另外的 1/4，然后是另外的 1/8，等等。所以大部分数组元素根本不需要查找，这节省了很多时间。直观来看，二分查找算法是非常快的。19.15

但它到底有多快？统计比较次数能提供算法效率的定量分析。为了明白算法在最差情况下的动作，现在来统计在含 n 个项的数组中进行查找时的最大比较次数。每次比较时，算法将数组一分为二。一次划分后，只剩下一半的项待查找。所以，开始时是 n 个项，然后留下  $n/2$  个项，然后是  $n/4$  个项，以此类推。最差情况下，查找持续到仅剩一项的时候。即有某

个正整数  $k, n/2^k$  将等于 1。值  $k$  就是数组分半的次数，或是递归调用 `binarySearch` 的次数。

如果  $n$  是 2 的幂次，对某个正整数  $k, n$  为  $2^k$ 。由对数的定义知， $k$  为  $\log_2 n$ 。如果  $n$  不是 2 的幂次，则可以找到一个正整数  $k$ ，让  $n$  介于  $2^{k-1}$  和  $2^k$  之间。例如，如果  $n$  是 14，则  $2^3 < 14 < 2^4$ 。所以，对某个  $k \geq 1$  有

$$\begin{aligned} 2^{k-1} &< n < 2^k \\ k-1 &< \log_2 n < k \\ k = 1 + \log_2 n &\text{ 向下取整} \\ &= \log_2 n \text{ 向上取整} \end{aligned}$$

简言之，

当  $n$  是 2 的幂次时， $k = \log_2 n$

当  $n$  不是 2 的幂次时， $k = \lceil \log_2 n \rceil$

一般的，`binarySearch` 的递归调用次数  $k$  是  $\lceil \log_2 n \rceil$ 。

### 注：天花板和地板

数  $x$  的天花板，表示为  $\lceil x \rceil$ ，是大于等于  $x$  的最小整数。例如， $\lceil 4.1 \rceil$  是 5。数  $x$  的地板，表示为  $\lfloor x \rfloor$ ，是小于等于  $x$  的最大整数。例如， $\lfloor 4.9 \rfloor$  是 4。当将一个正实数截断为一个整数时，实际上是忽略了其小数部分而计算的数的地板。

除最后一次外，每次调用 `binarySearch` 时都在目标与数组中间位置值之间进行两次比较：一次是测试相等，另一次是看小于或大于。所以二分查找最多执行  $2 \times \lceil \log_2 n \rceil$  次比较，所以最差情况下是  $O(\log n)$  的。

要查找 1000 个元素的数组，最差情况下，二分查找在目标与数组元素之间进行大约 10 次的比较。相反，简单顺序查找最多可能将目标与所有 1000 个数组项进行比较，平均将比较大约 500 个数组项。

### 注：数组中的二分查找的时间效率

最优情况： $O(1)$

最差情况： $O(\log n)$

平均情况： $O(\log n)$



**学习问题 9** 考虑 1 ~ 100 万之间的一个数。当我要猜你这个数时，告诉我我的猜测是正确的、太大还是太小。在正确猜到之前最多进行多少次尝试？提示：你要算猜测的次数，而不是比较的次数。

19.16

效率的另一种分析法。二分查找中每次比较都会找到数组的中间点。所以，为了在  $n$  个项中进行查找，二分查找法先查看中间项，然后在  $n/2$  个项中进行查找。如果令  $t(n)$  表示查找  $n$  个项时所需的时间，则最坏情况下有

$$\begin{aligned} t(n) &= 1 + t(n/2) \quad \text{对于 } n \geq 2 \\ t(1) &= 1 \end{aligned}$$

在第 9 章段 9.25 中见过这个递推关系。故有

$$t(n) = 1 + \log_2 n$$

所以，二分查找最差情况是  $O(\log n)$  的。

## 在无序链表中查找

ADT 线性表或 ADT 有序表的链式实现中，方法 `contains` 都在结点链表中查找目标。19.17  
你马上就会明白，顺序查找确实是唯一可行的选择。我们先讨论数据无序的链表，这是 ADT 线性表的通常情形。

不论线性表如何实现，线性表上的顺序查找要查看表中一连串的项，从第一项开始，直到找到所要找的项，或是查看了所有项后但没有成功。当采用链式实现时，从一个结点移到另一个结点，不像在数组中从一个位置移到另一个位置那样简单。尽管如此，仍可用迭代方式或递归方式实现对结点链表的顺序查找，且与数组中顺序查找的效率是一样的。

### 无序链表中的迭代顺序查找

含有线性表中各项的结点链表如图 19-7 所示。回忆第 12 章段 12.8 中，`firstNode` 是实现线性表的类的数据域。显然，方法可以使用引用 `firstNode` 来访问链表中的首结点，那它如何访问后面的结点呢？因为 `firstNode` 是永远指向链表中第一个结点的数据域，我们不能让查找过程改变它或是影响到线性表的其他方面。所以迭代方法 `contains` 应该使用局部引用变量 `currentNode`，初始时含有与 `firstNode` 相同的引用。执行下面的语句后，`currentNode` 指向下一个结点：

```
currentNode = currentNode.getNextNode();
```

迭代方式的顺序查找有下列简单的实现：

```
public boolean contains(T anEntry)
{
 boolean found = false;
 Node currentNode = firstNode;
 while (!found && (currentNode != null))
 {
 if (anEntry.equals(currentNode.getData()))
 found = true;
 else
 currentNode = currentNode.getNextNode();
 } // end while
 return found;
} // end contains
```

这个实现很像第 12 章 12.3 节中所给的实现。

### 无序链表中的递归顺序查找

当使用递归方式时，顺序查找查看线性表中的第一项，如果它不是要找的项，则查找线性表的其余部分。无论是仅使用线性表的 ADT 操作将查找实现为客户层的——如学习问题 4 中所做的，还是作为基于数组实现的线性表的一个公有方法——很像是我们在段 19.6 中所做的，这个递归方法都是一样的。我们将这同一个方法用于线性表的链式实现中，如下所示。

当线性表项保存在结点链表中时，如何实现“查找线性表的其余部分”这一步骤呢？前一段中见到的 `contains` 方法的迭代版本中，使用了一个局部变量 `currentNode` 在结点间移动。递归方法不能让 `currentNode` 作为局部变量，因为在每次递归调用时 `currentNode`

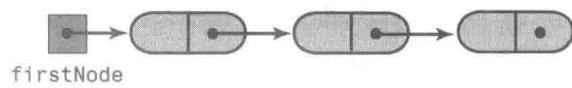


图 19-7 含有线性表中各项的结点链表

应该重置为初值。这样的方法需要将 `currentNode` 作为形参。但另一方面，我们会有另一个其形参依赖于线性表的实现方式的方法，这使得这个方法不能成为公有方法。就像我们之前在段 19.6 和段 19.13 中所做的一样，将这个查找方法作为私有的，且让一个公有方法调用它。

19.20

私有递归方法 `search` 检查形参 `currentNode` 所指向的结点中的线性表项。如果项不是所要找的，则方法递归调用自身，所带实参是指向链表中下一个结点的引用。所以方法 `search` 的实现如下。

```
// Recursively searches a chain of nodes for desiredItem,
// beginning with the node that currentNode references.
private boolean search(Node currentNode, T desiredItem)
{
 boolean found;
 if (currentNode == null)
 found = false;
 else if (desiredItem.equals(currentNode.getData()))
 found = true;
 else
 found = search(currentNode.getNextNode(), desiredItem);
 return found;
} // end search
```

现在，公有方法 `contains` 如下所示。

```
public boolean contains(T anEntry)
{
 return search(firstNode, anEntry);
} // end contains
```

注意到，调用方法 `search` 时，将形参 `currentNode` 的初值设置为 `firstNode`，类似于迭代方法中将局部变量 `currentNode` 设置为 `firstNode`。

## 链表中顺序查找的效率

19.21

链表上顺序查找的效率实际上与数组上的顺序查找一样。最优情况下，要找的项位于链表的第一个位置。所以最优查找将是  $O(1)$  的，因为你只需进行一次比较。最差情况下，要查找链表的所有项，对含  $n$  个结点的链表来说，要进行  $n$  次比较。所以，最差情况下顺序查找是  $O(n)$  的。一般地，要查看链表中大约一半的结点。所以平均情况下，查找过程是  $O(n/2)$  的，这同样是  $O(n)$  的。



**注：结点链表中的顺序查找的时间效率**

最优情况： $O(1)$

最差情况： $O(n)$

平均情况： $O(n)$

## 在有序链表中查找

现在在数据有序的链表中进行查找。ADT 有序表的链式实现中会出现这样的链表。

### 有序链表中的顺序查找

19.22

在数据有序的结点链表中进行查找，类似于在段 19.8 中描述的有序数组中的查找。这里，将那些逻辑用在如下的 `contains` 的实现中。

```

public boolean contains(T anEntry)
{
 Node currentNode = firstNode;
 while ((currentNode != null) &&
 (anEntry.compareTo(currentNode.getData()) > 0))
 {
 currentNode = currentNode.getNextNode();
 } // end while
 return (currentNode != null) && anEntry.equals(currentNode.getData());
} // end contains

```

方法遍历链表，直到到达一个含有要找对象的结点，或者检查了所有的结点但都没有成功时为止。为了得出结论，遍历过程之后的最后一个测试是必要的。因为数据是有序的，故判定 `anEntry` 不在链表中所花的时间，比数据无序时要少很多。

## 有序链表中的二分查找

数组上的二分查找先查看数组中间或接近中间位置的元素。通过计算 `first + (last - first)/2`，很容易确定这个元素的下标 `mid`，其中 `first` 和 `last` 分别是数组中第一个和最后一个元素的下标。访问这个中间元素也是容易的：对于数组 `a`，它就是 `a[mid]`。19.23

现在考虑之前在图 19-7 中见过的结点链表中的查找，链表中结点是有序的。你如何访问中间结点中的项？因为这个链表仅有 3 个结点，故你可以很快找到中间结点，但如果链表中含有 1000 个结点时怎么办？一般地，必须从第一个结点开始遍历链表，直到到达中间结点时为止。你如何知道何时到达那里？如果知道链表的长度，则可以将长度除以 2，计算出你要遍历的结点个数。细节不重要，只要能实现就可以，只是访问中间结点确实要费点力气。

查看完中间结点中的项后，可能需要忽略链表中的一半并查找另一半。忽略这部分链表时，不能改变这个链表。记住，你是要在链表中进行查找，而不是要毁掉它。一旦你知道了要查找哪一半，就必须再次遍历这个链表，找到它的中间结点。很显然，结点链表中的二分查找比顺序查找更难实现且效率更低。



注：结点链表中的二分查找是不现实的。

## 查找方法的选择

选择顺序查找还是选择二分查找。你刚看到了，应该使用顺序查找来查找结点链表。但如果想查找对象数组，需要知道哪个算法是可用的。要使用顺序查找，对象必须有方法 `equals`，用它来确定两个不同对象在某种意义上是否相等。因为所有对象都从 `Object` 类继承了 `equals` 方法，所以你必须保证查找的对象有重写的合适的 `equals` 版本。另一方面，要在对象数组中执行二分查找，对象必须有 `compareTo` 方法，且数组必须是有序的。如果这些条件不满足，则必须使用顺序查找。19.24

如果两个查找算法都能用于你的数组，应该使用哪个查找呢？如果数组很小，可以简单地使用顺序查找。如果数组很大且已有序，二分查找一般会比顺序查找快很多。但如果数组无序，你应该排序它然后再使用二分查找吗？答案要依赖于你查找数组的频繁程度。排序很费时间，往往多于顺序查找的时间。如果你只对无序数组查找很少的几次，那么对这个数组进行排序以便你能使用二分查找，似乎并不能节省时间，故应该使用顺序查找。

图 19-8 总结了顺序查找和二分查找的时间效率。只有顺序查找能适用于无序数据。二分查找的效率是用于有序数组的。对于一个大的有序数组，二分查找比顺序查找要快得多。

|            | 最优情况   | 平均情况        | 最差情况        |
|------------|--------|-------------|-------------|
| 顺序查找（无序数据） | $O(1)$ | $O(n)$      | $O(n)$      |
| 顺序查找（有序数据） | $O(1)$ | $O(n)$      | $O(n)$      |
| 二分查找（有序数据） | $O(1)$ | $O(\log n)$ | $O(\log n)$ |

图 19-8 查找的时间效率，用大  $O$  表示

19.25

**选择迭代查找还是选择递归查找。**因为递归顺序查找是尾递归，所以使用迭代查找时可以节省一些时间和空间。二分查找很快，所以递归实现时不需要很多额外的递归调用空间。另外，编写递归实现的二分查找的代码，也比编写迭代实现的代码更容易。试着编写迭代实现的二分查找代码，你就能明白这一点。（见本章结尾处的练习 6。）

## 本章小结

- 线性表、数组或是链表中的顺序查找，查看第一项、第二项，以此类推，直到或者找到具体的项，或者发现这个项没有出现在组内时为止。
- 顺序查找的平均情况的性能是  $O(n)$  的。
- 一般地迭代执行顺序查找，不过简单的递归方法也是可行的。
- 数组中的二分查找需要数组是有序的。它先查看所找的项是否出现在数组的中间位置。如果不是，查找决定项可能出现在数组的哪一半中，并在这一半上重复执行这个策略。
- 二分查找最坏情况下是  $O(\log n)$  的。
- 一般地递归执行二分查找，不过迭代方法也是可行的。
- 结点链表中的二分查找不现实。

## 程序设计技巧

- 实现了 `Comparable` 接口的类必须定义 `compareTo` 方法。这样的类还应该定义一个 `equals` 方法，它重写了继承于 `Object` 的 `equals` 方法。`compareTo` 和 `equals` 方法都应该使用相同的相等测试。段 19.13 中的 `binarySearch` 方法调用了 `equals` 和 `compareTo`。如果数组中的对象没有相应的 `equals` 方法，则 `binarySearch` 不会正确执行。但要注意，可以使用 `compareTo` 来替代 `equals` 进行相等测试。

## 练习

1. 修改段 19.6 给出的递归方法 `search`，它查看数组的最后一项而不是第一项。
2. 当顺序查找有序数组时，不需要查找整个数组就可以确定所给的项不在数组中出现。例如，如果你在数组 2 5 7 9 中查找 6，可以使用段 19.8 中描述的方法。即将 6 与 2 比较，然后与 5 比较，最后是与 7 相比。因为将 6 与 7 比较后没找到 6，所以不必继续查看，因为数组中的其他项都大于 7，不可能等于 6。因此不只简单地问 6 是否等于数组中的一项，还会问它是否大于一项。因为 6 大于 2，所以继续查找。对 5 也是如此。因为 6 小于 7，已经越过了数组中 6 应该会出现的地方，所以 6 不

在数组中。

- a. 写一个迭代方法 `inArray`, 在有序数组中进行顺序查找时, 利用所发现的这个特性。
- b. 写一个 `inArray` 可以调用的递归方法 `search`, 在有序数组中进行顺序查找时, 利用所发现的这个特性。
3. 前一个练习的问题 b 中描述的递归方法 `search`, 用来在图 19-6 所示的数组中查找 8 和 16 时, 分别进行多少次比较?
4. 跟踪段 19.13 给出的 `binarySearch` 方法, 在含下列值的数组中查找 4 的过程:  
5 8 10 13 15 20 22 26 30 31 34 40  
再跟踪查找 34 的过程。
5. 修改段 19.13 给出的 `binarySearch` 方法, 让它返回数组中第一个等于 `desiredItem` 的项的下标。如果数组中不包含这样的项, 则返回 `-(belongsAt+1)`, 其中 `belongsAt` 是 `desiredItem` 应该在数组中所处位置的下标。段 19.13 的最后, 学习问题 7 要求你在这种情况下返回 -1。注意, 当且仅当 `desiredItem` 没找到时, 方法的两种版本都返回一个负整数。
6. 实现迭代的数组中的二分查找。模仿段 19.13 给出这个方法的模型。
7. 写递归方法查找 `Comparable` 对象数组中的最大对象。与二分查找一样, 你的方法应该将数组一分为二。与二分查找不一样的是, 你的方法应该在两个子数组中都查找最大对象。数组中的最大对象应该是这两个最大对象中的较大者。
8. 假定你在可能含有重复值的无序对象数组中进行查找。修改算法, 返回数组中与所给对象相等的所有对象的下标列表。如果所给的对象不在线性表中, 则返回一个空表。
9. 对有序数组重复前一个练习。你的算法应该是递归的且是高效的。
10. 第 17 章讨论了 ADT 有序表。方法 `contains` 使用二分查找, 实现基于
  - a. 数组
  - b. 链表
 的有序表。

11. 考虑顺序查找最差情况下的比较次数  $f(n)$ 。

- a. 写出表示  $f(n)$  的递推关系。
- b. 对  $n$  进行归纳, 证明  $f(n)=n$ 。

12. 段 19.3 结尾处的学习问题 2 要求你仅使用 ADT 线性表的操作, 写一个对线性表执行顺序查找的迭代方法。比较这个方法与 ADT 操作 `contains` 的时间效率。
13. 在段 19.7 中, 我们说, 数组中的顺序查找平均情况下将检查  $n$  个项中的一半。让我们更仔细地看看这个计算。顺序查找或者成功或者失败。令  $\alpha$  是在数组中找到所找值的概率,  $1-\alpha$  则是找不到的概率。进一步假定, 这个值如果找到, 它在数组的每个位置的可能性是一样的。我们必须考虑每种可能性。

对每种情形, 我们计算比较次数并标注出其出现的概率。为找到查找中进行比较的平均次数, 先将每种情形的比较次数乘上每个概率。下表总结了这些结果。

|               | 概率         | 比较次数  | 乘积              |
|---------------|------------|-------|-----------------|
| 在下标 0 处找到     | $\alpha/n$ | 1     | $\alpha/n$      |
| 在下标 1 处找到     | $\alpha/n$ | 2     | $2\alpha/n$     |
| 在下标 2 处找到     | $\alpha/n$ | 3     | $3\alpha/n$     |
| ...           | ...        | ...   | ...             |
| 在下标 $n-2$ 处找到 | $\alpha/n$ | $n-1$ | $(n-1)\alpha/n$ |
| 在下标 $n-1$ 处找到 | $\alpha/n$ | $n$   | $\alpha$        |
| 没找到           | $1-\alpha$ | $n$   | $(1-\alpha)n$   |

- a. 将表中最后一列的所有乘积相加，计算平均比较次数。
  - b. 如果查找保证是成功的 ( $\alpha = 1$ )，则平均比较次数是多少？
  - c. 如果查找保证是失败的 ( $\alpha = 0$ )，则平均比较次数是多少？
  - d. 如果查找保证有一半是成功的 ( $\alpha = 0.5$ )，则平均比较次数是多少？
14. 重复前一个练习中的问题 a，但假定查找的项在数组每个位置出现的可能性是不一样的。将这  $n$  个项在数组中重排，越可能被查找的值越往前放。假定，一半的时间查找第一项， $1/4$  的时间查找第二项， $1/8$  的时间查找第三项，以此类推。有  $1/2^{n-1}$  的时间查找最后一项。据此修改前一个练习中的表。

## 项目

1. 插值查找 (interpolation search) 假定数组中的数据是有序的且均匀分布的。二分查找总是查看数组的中间位置项，而插值查找查看待查找项更可能出现的位置。例如，如果在有序的姓名目录中查找 Victoria Appleseed，你可能会查看靠近前面的位置而不是中间的位置。如果你发现有很多个 Appleseed，则你可能会在后面的 Appleseed 附近去查找 Victoria。

插值查找不是像二分查找那样总是去查看数组  $a$  中的元素  $a[mid]$ ，而是去检查元素  $a[index]$ ，其中

```
p = (desiredElement - a[first]) / (a[last] - a[first])
index = first + [(last - first) * p]
```

实现数组中的插值查找。对于特定的数组，比较插值查找和二分查找的结果。考虑数组的项均匀分布及不均匀分布的情况。

2. 当对象没有出现在数组中时，顺序查找这个对象必须检查整个数组。如果数组有序，则使用练习 2 中描述的方法可以改进这个查找。跳查 (jump search) 试图进一步减少比较的次数。

不是顺序检查数组  $a$  中的  $n$  个对象，而是对某个正数  $j < n$ ，查看元素  $a[j]$ ,  $a[2j]$ ,  $a[3j]$ , 等等。如果目标  $t$  小于这其中一个对象，只需检查位于当前对象和其前一个对象之间的部分数组元素。例如，如果  $t$  小于  $a[3j]$  但大于  $a[2j]$ ，则使用练习 2 中的方法查找元素  $a[2j+1]$ ,  $a[2j+1]$ , ...,  $a[3j-1]$ 。当  $t > a[k \times j]$ ，但  $(k+1) \times j > n$  时怎么办？

修改执行跳查的算法。 $j$  取值  $\lceil \sqrt{n} \rceil$ ，实现跳查算法。

3. 假定数值数据保存在二维数组中，例如图 19-9 所示的那样。每行和每列的数据递增有序。

- a. 为这种类型的数据设计一个高效的查找算法。
- b. 如果数组有  $m$  行  $n$  列，你设计的算法的大  $O$  性能是多少？
- c. 实现并测试你的算法。

|    |    |    |    |
|----|----|----|----|
| 1  | 4  | 55 | 88 |
| 7  | 15 | 61 | 91 |
| 14 | 89 | 90 | 99 |

图 19-9 用于项目 3 的二维数组

4. 考虑含  $n$  个有序数值的数组 `data` 和数值目标值表。你的目标是计算含有所有这些目标值的数组下标的最小范围。如果目标值小于 `data[0]`，范围应该从  $-1$  开始。如果目标值大于 `data[n-1]`，范围应该在  $n$  结束。

例如，所给数组如图 19-10 所示，目标值是 (8, 2, 9, 17)，则范围是  $-1 \sim 5$ 。

- a. 设计一个高效算法解决这个问题。

- b. 如果数组中有  $n$  个数据，表中有  $m$  个目标值，你算法的大  $O$  性能是多少？
- c. 实现并测试你的算法。

|   |   |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|
| 5 | 8 | 10 | 13 | 15 | 20 | 22 | 26 |
| 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  |

图 19-10 用于项目 4 的数组

- 5. 组织单词集合的一种方法是使用有序表的数组。数组为字母表中的每个字母保存一个有序表。要将一个单词添加到这个数据结构中，是将其添加到对应于单词首字母的有序表中。为这样的集合设计一个 ADT，包括操作 `add` 和 `contains`。为你的 ADT 设计一个 Java 接口。然后用一个类来实现这个接口并测试它。将一个文本文件中的单词添加到该数据结构中。
- 6. 写一个程序，从文本文件中读入一个 Java 程序，并执行下列任务。
  - a. 按字典序显示程序中用到的 Java 关键字列表。
  - b. 再做问题 a，但添加每个关键字在程序中出现的次数。
  - c. 再做问题 a 和 b，但添加包含关键字的行的行号。
- 7. 设计并实现一个查找单词字符串的算法，用来查找
  - a. 给定的单词。
  - b. 给定的短语，即一串单词。
- 8. 重做前一个问题中的问题 a，但这次将字符串中的单词排序，并标出每个单词在原字符串中的位置。这个查找算法与项目 7a 中的查找算法相比，速度如何？

## Java 插曲 8 |

Data Structures and Abstractions with Java, Fifth Edition

# 三论泛型

先修章节：附录 B、Java 插曲 5

本简短插曲描述如何在一个类或接口内定义并使用多个泛型数据类型。这个话题对于下一章要开始的 ADT 字典的讨论很重要。

## 多个泛型

J8.1 回忆 Java 插曲 1 的程序清单 JI1-2 中的类 OrderedPair：

```
public class OrderedPair<T> implements Pairable<T>
{
 private T first, second;

 <构造方法和方法getFirst, getSecond, toString及changeOrder
 的代码>

} // end OrderedPair
```

OrderedPair 实例中的两个配对的对象有相同的数据类型或因继承相关的数据类型。例如，下列语句将两个字符串配对创建了 OrderedPair 的一个对象：

```
OrderedPair<String> fruit = new OrderedPair<>("apples", "oranges");
```

J8.2 可以在一个类定义中定义多个泛型，方法是在类名后面的尖括号中写出用逗号分隔的标识符，如程序清单 JI8-1 所示的类 Pair 中那样。本例中，S 和 T 都是泛型数据类型。每一个都表示当实例化类的一个对象时由客户指定的一种实际的数据类型。

### 程序清单 JI8-1 类 Pair

```
1 public class Pair<S, T>
2 {
3 private S first;
4 private T second;
5
6 public Pair(S firstItem, T secondItem)
7 {
8 first = firstItem;
9 second = secondItem;
10 } // end constructor
11
12 public String toString()
13 {
14 return "(" + first + ", " + second + ")";
15 } // end toString
16 } // end Pair
```

例如，可以使用类 Pair，写下列语句将名字和电话号码组对，其中类 Name 由附录 B 的程序清单 B-1 给出：

```
Name joe = new Name("Joe", "Java");
String joePhone = "(401) 555-1234";
```

```
Pair<Name, String> joeEntry = new Pair<>(joe, joePhone);
System.out.println(joeEntry);
```

显示的输出是

(Joe Java, (401) 555-1234)



**学习问题 1** 你能使用程序清单 JI1-2 中定义的类 `OrderedPair`, 来配对两个有不同且无关数据类型的对象吗? 请解释原因。

**学习问题 2** 你能使用前一段中定义的类 `Pair` 来配对两个有相同数据类型的对象吗? 请解释原因。

**学习问题 3** 使用附录 B 中定义的类 `Name`, 写语句, 配对两位学生作为实验室合作伙伴。

**学习问题 4** 使用附录 B 中定义的类 `Name`, 写语句, 将你的名字与一个 `int` 类型变量 `number` 中保存的随机序列号配对。

## 第 20 章 |

Data Structures and Abstractions with Java, Fifth Edition

# 字 典

先修章节：第 10 章、Java 插曲 4、第 13 章、第 19 章、Java 插曲 8

### 目标

学习完本章后，应该能够

- 描述 ADT 字典的操作
- 区分字典和线性表
- 在程序中使用字典

如果需要查一个单词的意思，会在字典中查找；如果想找一位朋友的地址，会在地址簿里查找；如果想知道某人的电话号码，会在手机的联系人列表中查找，或者在线查找。

这里的每个例子都用到一种字典。本章描述并使用一种能概括日常使用的字典概念的抽象数据类型。后面的章节将研究这个 ADT 的实现。

前面的例子——查找单词的定义、朋友的地址或某人的电话号码，都是查找字典的例子。第 19 章讨论了如何在数组、结点链表，最根本的是线性表中进行查找。你会看到，字典提供了比线性表更强大的方法来组织可查找的数据。

## ADT 字典的规范说明

20.1

ADT 字典 (dictionary) 也称为映射 (map)、表 (table) 或关联数组 (associative array)，包含由两部分构成的项：

- 关键字，常称为查找键 (search key)，如英语单词或是人名
- 与键对应的值，如定义、地址或电话号码

查找键能让你定位到要找的项。

图 20-1 是一本普通的英语字典。每个项有一个作为查找键的单词及单词定义，后者是对应于前者的值。一般地，ADT 字典中的查找键和值是对象，如图 20-2 所示。每个查找键都与其所相关联的值配对。



图 20-1 一本英语字典

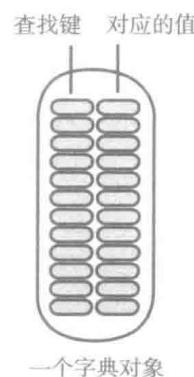


图 20-2 有查找键及匹配的对应值的 ADT 字典的实例

ADT 字典按查找键来组织及识别项，而不是按其他的准则，例如位置。所以，仅给出项的查找键，就可以从字典中获取或删除项。实际上，字典中的每个项都有一个查找键，这使得字典与线性表这样的 ADT 是有区别的。虽然你确实可以将有查找键的项放在线性表中，但线性表中的数据是按位置组织的，不是按查找键组织的。

有些字典有互异的查找键，但另外一些允许两个或多个项有相同的查找键。例如，按学生身份证号码组织的学生记录字典有唯一的查找键，因为这些号码是唯一的。另一方面，英语字典有重复的查找键，因为一个单词常常有几个意思。例如，我的字典中单词“book”有 3 个项：一个是名词，一个是动词，还有一个是形容词。

印刷版的自然语言字典、电话号码簿、图书目录及同义词词典，其中的项都是按查找键排序的。这些数据库都是字典，但 ADT 字典不要求项必须是有序的。有些字典按查找键将项进行排序，而有些字典的项是无序的。为什么印刷版字典的项要排序呢？因为排序能让读者更容易找到一个具体的项。相反，如果在计算机中的同义词典里查找一个单词的同义词，你不会知道项的次序。你也不会在乎项的次序，只要你能获取一个特定项就可以了。所以，相对于字典所必须具备的特性来说，字典是按查找键有序还是无序，仅仅只是实现细节。但要记住，任何实现细节都会以不同的方式影响 ADT 操作的效率。

ADT 字典的主要操作有插入、删除、获取、查找及遍历，不管具体实现中是按项排序或是允许查找键重复，这些操作都是大多数数据库和其他 ADT 所共有的。具体来说，这些操作是：

- 将含给定查找键及对应值的新项添加到字典中
- 删除与给定查找键对应的项
- 获取与给定查找键对应的值
- 查看字典中是否含有给定的查找键
- 遍历字典中所有的查找键
- 遍历字典中所有的值

此外，ADT 字典还有下列通常会含在 ADT 中的基本操作：

- 判定字典是否为空
- 获取字典中的项数
- 从字典中删除所有的项

 注：ADT 字典包含的项是按其查找键组织的键-值对。可以添加一个新项，给定查找键时，也可以获取或删除一个项，或者获知一个项是不是存在。另外，可以遍历字典的查找键或值。

#### 设计决策：字典项的查找键或值中应该含有 null 吗？

第 7 章段 7.19 的设计决策中讨论了哪些 ADT 中可以含有 null 数据。结论是，根据与项值无关的评判准则而决定项是无序或有序的任何一个 ADT，都可以含有 null 的项。所以到目前为止，大多数的 ADT 都允许 null 项。例如，ADT 线性表是基于位置的，所以 null 项可以作为一个项的占位符，这个项或是未来要插入的，或是仍为空的。使用这个方法，线性表中其他项的位置不会改变。例如，假定 20 匹马参加比赛并分配了从 1 ~ 20 的起跑位置。即使有一匹马没有进入起跑线而被剥夺了比赛资格，其他的马也不会改变位置。以类似的方式，栈、队列和双端队列都可以有 null

值，它们扮演占位符的角色。

优先队列和字典与基于位置的 ADT 存在很大差异。优先队列按照它的优先级对项进行排序，而这个是基于项本身的，所以不应该含有 null 项。字典保存键-值对。有序字典按照其查找键组织项。在无序字典中，虽然项没有按序保存，但要按照查找键来查找项。为简化我们的实现，我们决定不允许查找键是 null。

项的值部分应该是 null 吗？即不是 null 的查找键应该对应一个 null 值吗？我们可以将 null 解释为没有值，决定在我们的字典中不允许有键-null 对。如果我们阻止了有 null 值的项，那么字典方法就可以返回 null，以表示没有找到给定的查找键。另一种办法是允许有键-null 项。使用这个方法时，如果没有找到键，可以抛出 `KeyNotFoundException` 异常表示失败。我们是否想要保存一个其查找键非空但值为空的项？客户可能想维护特定的查找键，并让某些键与作为占位符的 null 值相对应。显然，允许或阻止查找键和值为 null 的决定，取决于具体的情况。这里我们的字典不允许查找键和值是 null。

下面的规范说明为 ADT 字典定义了一组可能的操作：

| 抽象数据类型：字典                       |     |                                                    |                                                                              |
|---------------------------------|-----|----------------------------------------------------|------------------------------------------------------------------------------|
| 数据                              |     |                                                    |                                                                              |
| 操作                              | 伪代码 | UML                                                | 描述                                                                           |
| <code>add(key, value)</code>    |     | <code>+add(key:K,value:V):void</code>              | 任务：将 (key,value) 对添加到字典中<br>输入：key 是查找键对象，value 是对应的对象<br>输出：无               |
| <code>remove(key)</code>        |     | <code>+remove(key:K):V</code>                      | 任务：从字典中删除对应于给定查找键的项<br>输入：key 是查找键对象<br>输出：返回对应于查找键的值；或者，如果这样的对象不存在，则返回 null |
| <code>getValue(key)</code>      |     | <code>+getValue(key:K):V</code>                    | 任务：从字典中获取与给定查找键对应的值<br>输入：key 是查找键对象<br>输出：返回对应于查找键的值；或者，如果这样的对象不存在，则返回 null |
| <code>contains(key)</code>      |     | <code>+contains(key:K):boolean</code>              | 任务：查看字典中是否存在含有给定查找键的项<br>输入：key 是查找键对象<br>输出：如果字典中的一个项含有与所给查找键一样的键，则返回真      |
| <code>getKeyIterator()</code>   |     | <code>+getKeyIterator():Iterator&lt;K&gt;</code>   | 任务：创建遍历字典中所有查找键的迭代器<br>输入：无<br>输出：返回一个迭代器，能顺序访问字典中的查找键                       |
| <code>getValueIterator()</code> |     | <code>+getValueIterator():Iterator&lt;V&gt;</code> | 任务：创建遍历字典中所有值的迭代器<br>输入：无<br>输出：返回一个迭代器，能顺序访问字典中的值                           |
| <code>isEmpty()</code>          |     | <code>+isEmpty():boolean</code>                    | 任务：查看字典是否为空<br>输入：无<br>输出：如果字典为空，则返回真                                        |

(续)

| 伪代码       | UML                | 描述                                         |
|-----------|--------------------|--------------------------------------------|
| getSize() | +getSize():integer | 任务：得到字典的大小<br>输入：无<br>输出：返回字典中当前项(键-值对)的个数 |
| clear()   | +clear():void      | 任务：删除字典中的所有项<br>输入：无<br>输出：无               |

细化规范说明。即使所有的字典都有这组常见的操作，你还是需要根据查找键在字典中是否唯一来细化某些规范说明。

20.3

- **唯一的查找键。**方法 add 能确保字典中的查找键是唯一的。如果 key 已在字典中，则操作 add(key, value) 或者拒绝添加另一个键-值项，或者将对应于 key 的值改为 value。后一种情形中，方法能够返回原来被替换掉的值，而不是像前面描述的那样没有输出。

不管 add 如何保证查找键的唯一性，其他方法的实现都比允许有重复查找键时的实现要简单。例如，方法 remove 和 getValue 或者找到对应于所给查找键的一个值，或者发现不存在这样的项。

- **重复的查找键。**如果方法 add 将每个给定的键-值项都添加到字典中，则方法 remove 和 getValue 必须处理有相同查找键的多个项。哪个项将被删除或返回？方法 remove 可以删除它能找到的对应于给定查找键的第一个值，也可以删除与其对应的所有值。方法 getValue 可以返回它找到的第一个值。或者，也可以修改 getValue 方法，比如让它返回一串值。

另一种可能的方法是，仅当多个项有相同的主查找键时，使用第二个查找键。例如，如果你打查号台查询一个常见名字的电话，比如约翰·史密斯，肯定要被询问约翰的地址。

为简单起见，我们假定查找键唯一，并在本章末的练习和项目中考虑重复查找键的情况。

## Java 接口

程序清单 20-1 含有用于 ADT 字典的接口，接口中规定查找键是唯一的。add 方法替换字典中已有的与查找键对应的值。注意，这个方法在最初的规范说明中不是 void 方法。

20.4

与用于 ADT 线性表和 ADT 有序表的接口一样，这个接口规范说明了通常情况下的项的数据类型。因为查找键的数据类型可能不同于相应值的数据类型，所以我们使用两个泛型参数 K 和 V。K 表示查找键的数据类型，而 V 表示相应值的类型。

### 程序清单 20-1 用于 ADT 字典的接口

```

1 import java.util.Iterator;
2 /**
3 An interface for a dictionary with distinct search keys.
4 Search keys and associated values are not null.
5 */
6 public interface DictionaryInterface<K, V>
7 {
8 /** Adds a new entry to this dictionary. If the given search key already
9 exists in the dictionary, replaces the corresponding value.
10 @param key An object search key of the new entry.
11 @param value An object associated with the search key.

```

```

12 @return Either null if the new entry was added to the dictionary
13 or the value that was associated with key if that value
14 was replaced. */
15 public V add(K key, V value);
16
17 /** Removes a specific entry from this dictionary.
18 @param key An object search key of the entry to be removed.
19 @return Either the value that was associated with the search key
20 or null if no such object exists. */
21 public V remove(K key);
22
23 /** Retrieves from this dictionary the value associated with a given
24 search key.
25 @param key An object search key of the entry to be retrieved.
26 @return Either the value that is associated with the search key
27 or null if no such object exists. */
28 public V getValue(K key);
29
30 /** Sees whether a specific entry is in this dictionary.
31 @param key An object search key of the desired entry.
32 @return True if key is associated with an entry in the dictionary. */
33 public boolean contains(K key);
34
35 /** Creates an iterator that traverses all search keys in this dictionary.
36 @return An iterator that provides sequential access to the search
37 keys in the dictionary. */
38 public Iterator<K> getKeyIterator();
39
40 /** Creates an iterator that traverses all values in this dictionary.
41 @return An iterator that provides sequential access to the values
42 in this dictionary. */
43 public Iterator<V> getValueIterator();
44
45 /** Sees whether this dictionary is empty.
46 @return True if the dictionary is empty. */
47 public boolean isEmpty();
48
49 /** Gets the size of this dictionary.
50 @return The number of entries (key-value pairs) currently
51 in the dictionary. */
52 public int getSize();
53
54 /** Removes all entries from this dictionary. */
55 public void clear();
56 } // end DictionaryInterface

```

20.5 下面来看看如何创建实现了 DictionaryInterface 的类 Dictionary 的实例。字典将含有某学校中学生的数据。假定学号是查找键，类 Student 表示学生数据。下列语句创建了实例 DataBase。

```
DictionaryInterface<String, Student> DataBase = new Dictionary<>();
```

String 对应于 DictionaryInterface 中的参数 K，所以接口中出现的每个 K 都用 String 来替换。类似地，Student 替换接口中出现的每个 V。这些实际的类型也要对应到类 Dictionary 中的泛型。

我们在本章后面会更详细地讨论字典的几个示例。

## 迭代器

20.6 方法 getKeyIterator 和 getValueIterator 都返回一个迭代器，它符合我们在 Java

插曲 4 中讨论的 `java.util.Iterator` 接口标准。可以用下列语句为前一段所示的字典 `dataBase` 创建迭代器：

```
Iterator<String> keyIterator = dataBase.getKeyIterator();
Iterator<Student> valueIterator = dataBase.getValueIterator();
```

回忆一下，`Iterator` 的定义中规范说明了泛型。这里，我们为 `String` 类型的查找键定义了一个迭代器，为 `Student` 类型的值定义了另一个迭代器。这些迭代器按照项在字典中出现的次序，依次遍历字典项。

你可以单独或同时使用这两个迭代器，如图 20-3 所示。即你可以：

- 使用 `keyIterator`，遍历字典中的所有查找键，但不遍历值；
- 使用 `valueIterator`，遍历所有的值，但不遍历查找键；
- 使用 `keyIterator` 和 `valueIterator`，同步遍历所有的查找键及所有的值。

在最后一种情形中，`keyIterator` 返回的第  $i$  个查找键对应于 `valueIterator` 返回的字典中第  $i$  个值。显然，这两个迭代器有相同的长度，因为字典中查找键的个数与值的个数相同。

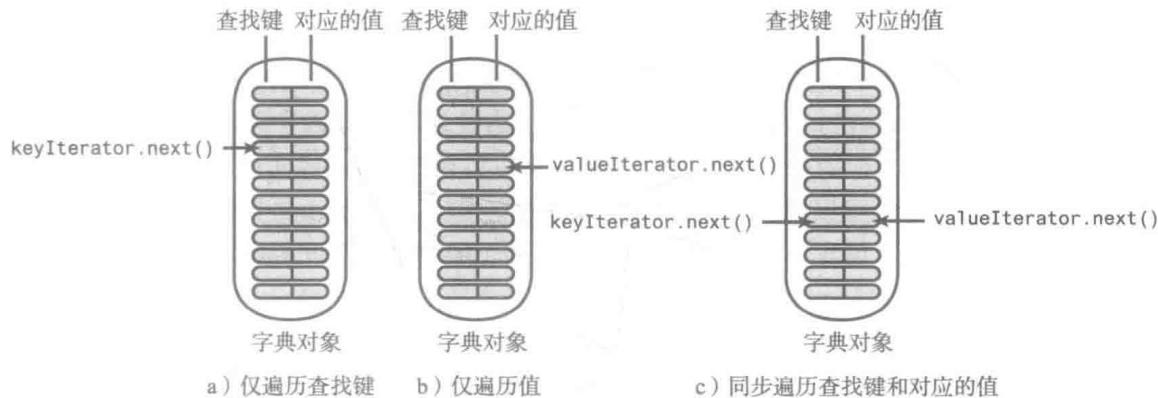


图 20-3 遍历字典中键和值的两个迭代器，既可独立又可同步

下面的循环按键 – 值对的形式显示字典中的每一项。

```
while (keyIterator.hasNext())
 System.out.println(keyIterator.next() + ", " + valueIterator.next());
```

对于有序字典，`keyIterator` 按序遍历查找键。对于无序字典，遍历的次序并不确定。下一节的例子将说明不同情况下的这些迭代器。



**注：**字典值的迭代对应于其查找键的迭代。即一个迭代中字典的第  $i$  个值，对应于另一个迭代中的第  $i$  个查找键。



**学习问题 1** 如果类 `Dictionary` 实现了 `DictionaryInterface`，写创建空字典 `myDictionary` 的 Java 语句。这个字典将含有你朋友的名字和电话号码。假定名字是查找键，用类 `Name` 来表示它们。每个电话号码都是一个字符串。

**学习问题 2** 写 Java 语句，将你的名字和电话号码添加到学习问题 1 中创建的字典中。

**学习问题 3** 写 Java 语句，如果 Britney Storm 在学习问题 1 描述的字典中，则显示她的电话号码，如果不在，显示一条错误信息。

## 使用 ADT 字典

本节的 3 个示例说明在程序中如何使用 ADT 字典。从创建电话号码簿开始。

### 问题求解：电话号码簿



电话号码簿包含生活在指定地区的人名和电话号码。实现一个软件来定义这样一个电话号码簿。

20.7

电话号码簿上最常见的操作是获取指定人名的电话号码。所以，使用 ADT 字典来表示电话号码簿是一个好选择。显然，名字应该是查找键，电话号码应该是对应的值。通常字典有序时获取电话号码的效率更高些，但也并不总是这样。另外，有序的字典更容易用来创建名字按字典序排列的印刷版号码簿。为简化这个例子，我们假定字典中所含的名字没有重复值。

主要的任务，至少是最初阶段的主要任务，是由可用的名字及电话号码创建字典。将这些数据保存在一个文本文件中更便于完成这项工作。在创建了电话号码簿后，对字典的操作，如添加一项、删除一项，或是修改一个电话号码，往往比查找给定的名字要少。对于将数据打印出来或是备份到一个文本文件中这样的操作，遍历字典都是重要的，但这个操作太不经常做了。正如第 4 章我们说明的，你应该基于预期用途的效率来选择 ADT 的实现。

20.8

**设计和使用类 TelephoneDirectory。**下一步是设计表示电话号码簿的类。用有序字典来表示含有名字 – 号码对的数据。每个人的名字可以是类 Name 的实例，这个类在附录 B 中遇到过，而电话号码可以是不带嵌入的空白符的字符串。图 20-4 是我们设计的类图。类 TelephoneDirectory 包含字典 phoneBook 的实例。类中有方法 readFile，它从文件读入数据，并将数据添加到 phoneBook 中。类中还有方法 getPhoneNumber，用来获取给定名字的电话号码。为简单起见，忽略前一段中提到的其他操作。

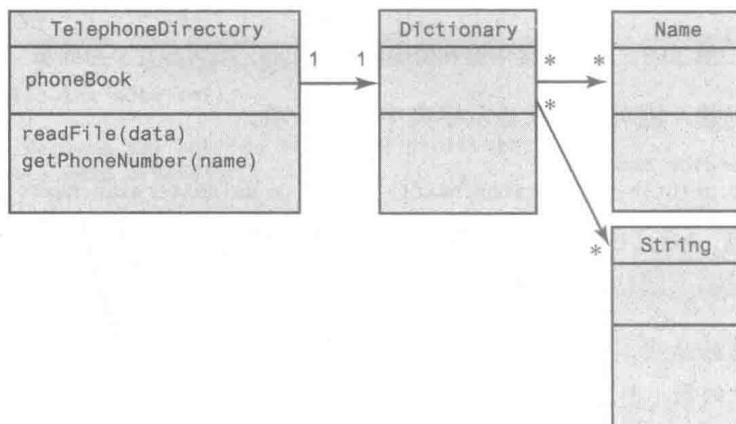


图 20-4 电话号码簿的类图

在实现类 TelephoneDirectory 之前，先考虑它的使用。客户应该创建 TelephoneDirectory 的实例，并调用 readFile 方法读入数据。在程序清单 20-2 所示的 main 方法中，前两个做了标记的行执行这两步。给定的文本文件的名字是 data.txt，main 将创建文件的扫描器，并将其传给 readFile。注意，创建扫描器时可能会遇到异常。如果需要更深入地了解异常或是文件，可分别参看 Java 插曲 2 和补充材料 2（在线）。

文件读入后，`main`方法通过私有方法`getName`与用户进行交互。从用户读入的每个名字传给方法`getPhoneNumber`，这个名字是在电话号码簿中进行查找的关键字。要注意，`getName`是如何用`Scanner`来读入用户的输入并进行处理的。

### 程序清单 20-2 类 TelephoneDirectory 的客户

```

1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4 public class Driver
5 {
6 private static final Name INPUT_ERROR = new Name("error", "error");
7 private static final Name QUIT = new Name("quit", "quit");
8
9 public static void main(String[] args)
10 {
11 TelephoneDirectory directory = new TelephoneDirectory();
12 String fileName = "data.txt"; // Or file name could be read
13
14 try
15 {
16 Scanner data = new Scanner(new File(fileName));
17 directory.readFile(data);
18 }
19 catch (FileNotFoundException e)
20 {
21 System.out.println("File not found: " + e.getMessage());
22 }
23
24 Name nextName = getName(); // Get name for search from user
25 while (!nextName.equals(QUIT))
26 {
27 if (nextName.equals(INPUT_ERROR))
28 System.out.println("Error in entering name. Try again.");
29 else
30 {
31 String phoneNumber = directory.getPhoneNumber(nextName);
32 if (phoneNumber == null)
33 System.out.println(nextName + " is not in the directory.");
34 else
35 System.out.println("The phone number for " + nextName +
36 " is " + phoneNumber);
37 } // end if
38
39 nextName = getName();
40 } // end while
41 System.out.println("Bye!");
42 } // end main
43
44 // Returns either the name read from user, INPUT_ERROR, or QUIT.
45 private static Name getName()
46 {
47 Name result = null;
48 Scanner keyboard = new Scanner(System.in);
49
50 System.out.print("Enter first name and last name, " +
51 "or quit to end: ");
52 String line = keyboard.nextLine();
53
54 if (line.trim().toLowerCase().equals("quit"))
55 result = QUIT;
56 else
57 {

```

```

58 String firstName = null;
59 String lastName = null;
60 Scanner scan = new Scanner(line);
61 if (scan.hasNext())
62 {
63 firstName = scan.next();
64 if (scan.hasNext())
65 lastName = scan.next();
66 else
67 result = INPUT_ERROR;
68 }
69 else
70 result = INPUT_ERROR;
71
72 if (result == null)
73 // First and last names have been read
74 result = new Name(firstName, lastName);
75 } // end if
76
77 return result;
78 } // end getName
79 } // end Driver

```

**输出**

```

Enter first name and last name or quit to end: Maria Lopez
The phone number for Maria Lopez is 401-555-1234
Enter first name and last name or quit to end: Hunter
Error in entering name. Try again.
Enter first name and last name or quit to end: Hunter Smith
Hunter Smith is not in the directory.
Enter first name and last name or quit to end: quit
Bye!

```

20.9

开始实现。类 TelephoneDirectory 的开头部分如程序清单 20-3 所示。假定类 SortedDictionary 实现了有序版本的 ADT 字典，且查找键唯一。有序字典需要查找键属于实现了接口 Comparable 的类。我们假定 Name 满足要求。

**程序清单 20-3** 类 TelephoneDirectory 的框架

```

1 import java.util.Iterator;
2 import java.util.Scanner;
3 public class TelephoneDirectory
4 {
5 private DictionaryInterface<Name, String> phoneBook;
6
7 public TelephoneDirectory()
8 {
9 phoneBook = new SortedDictionary<>();
10 } // end default constructor
11
12 /** Reads a text file of names and telephone numbers.
13 * @param data A text scanner for the text file of data. */
14 public void readFile(Scanner data)
15 {
16
17 . . . <See Segment 20.10. >
18
19 } // end readFile
20
21 /** Gets the phone number of a given person. */
22 public String getPhoneNumber(Name personName)
23 {
24

```

```

25 . . . <See Segment 20.11.>
26
27 } // end getPhoneNumber
28 ...

```

为实现方法 `readFile`，必须了解数据文件是什么样子的。假定文件中的每一行含有由空格分开的 3 个字符串——名字、姓和电话号码。20.10 所以一行典型的数据是这样的：

Suzanne Nouveaux 401-555-1234

方法 `readFile` 必须读入这些字符串中的每一个。回忆一下，程序清单 20-2 中的 `main` 方法，创建了用于文件的扫描器，并将它传给了 `readFile` 文件。使用 `Scanner` 的 `next` 方法，`readFile` 可以读入数据文件一行中的每个字符串，并将它们分别赋给变量 `firstName`、`lastName` 和 `phoneNumber`。然后，下列 Java 语句将所需的项添加到字典 `phoneBook` 中。

```

Name fullName = new Name(firstName, lastName);
phoneBook.add(fullName, phoneNumber);

```

我们假定文本文件含有的名字是不同的。

下面是 `readFile` 的定义。

```

public void readFile(Scanner data)
{
 while (data.hasNext())
 {
 String firstName = data.next();
 String lastName = data.next();
 String phoneNumber = data.next();

 Name fullName = new Name(firstName, lastName);
 phoneBook.add(fullName, phoneNumber);
 } // end while
 data.close();
} // end readFile

```

使用 `Scanner` 的方法 `hasNext` 和 `next`，可以从文本文件中获取作为字符串的每个名字和电话号码。然后，使用前面提到的两条语句，创建一个 `Name` 对象，并将其和电话号码一起添加到字典中。



### 程序设计技巧：java.util.Scanner

类 `Scanner` 能将字符串分隔为子串，或称为记号 (token)，它们由称为分隔符 (delimiter) 的符号分隔开。默认情况下，分隔符是空格。可以将待分析的字符串传给 `Scanner` 的构造方法，或是将表示为 `java.io.File` 实例的一个文本文件传给它。

`Scanner` 类中的下列方法能从任何字符串中提取记号。

```

public String next();
public boolean hasNext();

```

补充材料 1 (在线) 中从段 S1.81 开始详细讨论了 `Scanner`。



### 学习问题 4 虽然语句

```
directory.readFile(data);
```

写在程序清单 20-2 中 `main` 方法开头附近的 `try` 块内，但它不一定非得在那里。解释它出现的位置，为什么它能出现在 `try` 块外，而且做什么才能够移动它。

**20.11** **查找方法。**类 TelephoneDirectory 有一个查找某人电话号码的方法。这个方法需要一个人的名字，它必须由用户提供。如果假定客户程序与用户交互，并且用户为方法提供了所需的名字——和程序清单 20-2 的客户程序所做的一样——则我们定义的方法如下：

```
public String getPhoneNumber(Name personName)
{
 return phoneBook.getValue(personName);
} // end getPhoneNumber
```

方法返回含有所需电话号码的字符串，如果没找到号码则方法返回 null。

我们还可以定义一个类似的方法，替代前一个方法，或是作为它的附加，如下所示。

```
public String getPhoneNumber(String firstName, String lastName)
{
 Name fullName = new Name(firstName, lastName);
 return phoneBook.getValue(fullName);
} // end getPhoneNumber
```

添加或删除一个人或修改一个人的电话号码等其他方法都很简单，留作练习。



**学习问题 5** 为类 TelephoneDirectory 实现从字典中删除一个项的方法。给定人名，方法将返回这个人的电话号码，如果这个人不在字典中则返回 null。

**学习问题 6** 为类 TelephoneDirectory 实现修改一个人的电话号码的方法。给定人名，方法将返回这个人原来的电话号码，如果这个人不在字典中则返回 null，并将项添加到字典中。

## 问题求解：单词的频率



有些字处理程序提供了文档中每个单词出现的次数。创建类 FrequencyCounter 提供这个功能。

**20.12**

这个类有些像前一个例子中的类，所以我们省略一些设计细节。总的来说，当从文本文件读入文档时，类需要对单词的每次出现进行计数。然后显示结果。例如，如果文本文件含有 row, row, row your boat

则期望的输出将是

```
boat 1
row 3
your 1
```

这个类要有一个构造方法及方法 `readFile` 和 `display`。与前一个例子一样，`readFile` 方法将从文件中读入输入的文本。然后 `display` 方法将显出输出结果。程序清单 20-4 所示为 `FrequencyCounter` 的客户程序。它类似于前一个例子中客户程序的开头部分。

### 程序清单 20-4 FrequencyCounter 类的客户程序

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4
5 public class Driver
6 {
7 public static void main(String[] args)
```

```

8 {
9 FrequencyCounter wordCounter = new FrequencyCounter();
10 String fileName = "Data.txt"; // Or file name could be read
11
12 try
13 {
14 Scanner data = new Scanner(new File(fileName));
15 wordCounter.readFile(data);
16 }
17 catch (FileNotFoundException e)
18 {
19 System.out.println("File not found: " + e.getMessage());
20 }
21 wordCounter.display();
22 } // end main
23 } // end Driver

```

输出

```

boat 1
row 3
your 1

```

ADT 字典适用于这个问题吗？文档中的单词及它出现的频度组成一对，它符合字典中的项。如果我们想知道给定单词的频度，则单词应该是查找键。另外，字典中的单词必须是唯一的，如果它们是有序的，则可以按字典序来显示它们。所以对本程序来讲，查找键唯一的有序字典是合适的选择。与前一个例子一样，我们假定 `SortedDictionary` 的实现满足要求。

20.13

字典将是新类 `FrequencyCounter` 的数据域，这个类的开头很像是前一个例子中的类 `TelephoneDirectory`。我们称本例中使用的字典为 `wordTable`。因为任何字典项的值部分都是一个对象，所以我们使用包装类 `Integer` 来表示每个频度。由此得到类的开头部分，列在程序清单 20-5 中。

### 程序清单 20-5 类 FrequencyCounter 的框架

```

1 import java.util.Iterator;
2 import java.util.Scanner;
3 public class FrequencyCounter
4 {
5 private DictionaryInterface<String, Integer> wordTable;
6
7 public FrequencyCounter()
8 {
9 wordTable = new SortedDictionary<>();
10 } // end default constructor
11
12 /** Reads a text file of words; counts their frequencies of occurrence.
13 * @param data A text scanner for the text file of data. */
14 public void readFile(Scanner data)
15 {
16
17 . . . <See Segment 20.16.>
18
19 } // end readFile
20
21 /** Displays words and their frequencies of occurrence. */
22 public void display()
23 {
24

```

```

25 . . . <See Segment 20.17.>
26
27 } // end display
28 } // end FrequencyCounter

```

20.14

**创建字典。**现在来看看方法 `readFile`, 它从文本文件创建字典。调用这个方法的方式与之前在程序清单 20-4 中是一样的。即客户程序将与文本文件对应的 `Scanner` 对象传给 `readFile`。然后方法使用 `Scanner` 类的方法 `hasNext` 和 `next` 处理文本文件, 这与段 20.10 中 `readFile` 处理文件的方式是一样的。

从文件中提取下一个单词之后, `readFile` 检查这个单词是否在字典中。如果不在, 将其与 1 一起添加。即这个单词到目前为止仅出现一次。但是, 如果单词已在字典中, 则获取它的值——它的计数——且加 1, 然后再存回字典中。为避免大小写问题, `readFile` 可以将读入的所有单词都改为小写。

20.15

**分隔符。**段 20.10 结尾处的“程序设计技巧”提到, 默认情况下, `Scanner` 使用空格当作分隔符。但如段 20.12 所给的示例一样, 数据中可能含有标点符号, 所以这些符号也必须是分隔符。你可以使用 `Scanner` 的方法 `useDelimiter` 来指定分隔符。使用补充材料 1 (在线) 中图 S1-6 所示的符号表示它们。(见段 S1.83。) 然后将分隔符的字符串传给 `useDelimiter`。

指定空格和标点符号作为分隔符的最简单方法是使用符号 `\W`, 因为它表示除字母、数字或下划线之外的任何符号。然后将 `useDelimiter` 的实参写为 “`\W+`”。记住, 必须使用双斜杠来区分记号和转义字符。加号 + 表示一次或多次出现。所以语句

```
dataFile.useDelimiter("\W+");
```

将分隔符设置为标点符号、空格或数据中没有出现的其他符号的一次或多次出现。

**!** **程序设计技巧:** 当使用 `Scanner` 对象来处理文本时, 任何没有出现在所需记号中的字符都可以是分隔符。你可以使用一个特殊的记号来创建这些分隔符串, 并将它传给 `Scanner` 的方法 `useDelimiter`。更多细节请参看补充材料 1 (在线) 中的段 S1.82。

20.16

前面讨论的内容体现在 `readFile` 方法的如下实现中。

```

/** Reads a text file of words and counts their frequencies of occurrence
 * @param data A text scanner for the text file of data. */
public void readFile(Scanner data)
{
 data.useDelimiter("\W+");
 while (data.hasNext())
 {
 String nextWord = data.next();
 nextWord = nextWord.toLowerCase();
 Integer frequency = wordTable.getValue(nextWord);

 if (frequency == null)
 { // Add new word to table
 wordTable.add(nextWord, Integer.valueOf(1));
 }
 else
 { // Increment count of existing word; replace wordTable entry
 frequency++;
 wordTable.add(nextWord, frequency);
 } // end if
 } // end while
}

```

```
 data.close();
} // end readFile
```



**学习问题 7** 前一个方法 `readFile` 没有调用 `contains` 方法来查看单词是否已在字典中，而是调用了 `getValue`。为什么这样做？

显示字典。现在已经创建了字典，我们需要显示结果。对查找键的迭代将得到按字典序给出的单词。对值的同步迭代能得到相应的频度。下列方法完成了这个任务。

```
public void display()
{
 Iterator<String> keyIterator = wordTable.getKeyIterator();
 Iterator<Integer> valueIterator = wordTable.getValueIterator();
 while (keyIterator.hasNext())
 {
 System.out.println(keyIterator.next() + " " + valueIterator.next());
 } // end while
} // end display
```



**学习问题 8** 为类 `FrequencyCounter` 实现第二个 `display` 方法，方法的唯一形参是频度，对于给定的频度，方法仅显示具有这个频度的单词。

## 问题求解：单词的索引



索引（index）提供了在大型文档内找到某些单词出现位置的一种方法。例如，本书的索引是按字典序排列的单词及其出现的页码组成的值对的列表。对于本问题，我们将为文本文件中的所有单词创建更简单的一类索引——称为词语索引（condordance）。索引不是提供某个单词所在的页码，而是提供其所在的行号。

我们先来看看索引的一个例子。假定文本文件仅含有下面这些行：

Learning without thought is labor lost;  
thought without learning is perilous.

文件中所有单词的下列索引指明这个单词出现在哪行中：

```
is 1 2
labor 1
learning 1 2
lost 1
perilous 2
thought 1 2
without 1 2
```

虽然一个单词可能出现在文件的多个行中，但它在索引中仅出现一次。类似于前一个单词频度的示例，索引的这个特性暗示我们要使用字典，其查找键是索引中的单词。但与单词频度示例不同的是，这些单词对应的值是一个行号的列表。因为行号是有序的，故可以使用ADT有序表。不过，处理文件中的行是按序进行的，所以可以将行号添加到普通无序表的结尾，从而得到有序结果。

20.19

用来表示索引的类 `Concordance` 与前一个示例中得到的类 `FrequencyCounter`，它们的设计与实现都十分相似。事实上，这两个类的使用几乎是相同的。使用 `Concordance` 替换程序清单 20-4 中的 `FrequencyCounter`，可得到 `Concordance` 的客户程序。

程序清单 20-6 是类 `Concordance` 的框架。注意它与段 20.13 的程序清单 20-5 所给的 `FrequencyCounter` 的相似之处。除了方法的实现之外，主要的区别是每个字典项的值的数据类型不同。因为值是 `Integer` 对象的线性表，又因为我们想遍历每个线性表来显示行号，所以值的数据类型是 `ListWithIteratorInterface<Integer>`。第 13 章段 13.8 中定义了这个接口，接口中有方法 `iterator` 和 `getIterator`，还有 `ListInterface` 中的方法。

### 程序清单 20-6 类 `Concordance` 的框架

```

1 import java.util.Iterator;
2 import java.util.Scanner;
3
4 public class Concordance
5 {
6 private DictionaryInterface<String, ListWithIteratorInterface<Integer>>
7 wordTable;
8
9 public Concordance()
10 {
11 wordTable = new SortedDictionary<>();
12 } // end default constructor
13
14 /** Reads a text file of words and creates a concordance.
15 * @param data A text scanner for the text file of data. */
16 public void readfile(Scanner data)
17 {
18
19 . . . <See Segment 20.20.>
20
21 } // end readfile
22
23 /** Displays words and the lines in which they occur. */
24 public void display()
25 {
26
27 . . . <See Segment 20.21.>
28
29 } // end display
30 } // end Concordance

```

20.20

方法 `readFile`。方法 `readFile` 读入文本文件，并用字典 `wordTable` 来创建索引。因为必须记录下每个单词的行号，所以每次读入文件中的一行。在移到下一行之前要处理该行中的所有单词。故下面实现的 `readFile` 中含有嵌套的两个循环。外层循环使用作为实参传入的 `Scanner` 对象从文件中读入各行。内层循环使用另一个扫描器，一旦读入一行，就从中提取各单词。第 13 章段 13.9 中的类 `LinkedListWithIterator` 用来生成每个行号线性表。

```

public void readFile(Scanner data)
{
 int lineNumber = 1;
 while (data.hasNext())
 {
 String line = data.nextLine();
 line = line.toLowerCase();
 Scanner lineProcessor = new Scanner(line);

```

```

lineProcessor.useDelimiter("\\W+");
while (lineProcessor.hasNext())
{
 String nextWord = lineProcessor.next();
 ListWithIteratorInterface<Integer> lineList =
 wordTable.getValue(nextWord);

 if (lineList == null)
 { // Create new list for new word; add word and list to index
 lineList = new LinkedListWithIterator<>();
 wordTable.add(nextWord, lineList);
 } // end if

 // Add line number to end of list so list is sorted
 lineList.add(lineNumber);
} // end while
lineNumber++;
} // end while
data.close();
} // end readFile

```

这个方法最令人感兴趣的部分，是对应于查找键的值组成的行号线性表。因为我们选择线性表的链式实现，所以必须考虑添加到线性表表尾的效率。如果底层的结点链表仅有一个指向首结点的引用——如 `LinkedListWithIterator` 中这样——则每次添加都需要遍历到链表表尾。选择带链尾结点引用的线性表的实现，可使在链表表尾添加的效率更高。我们在第 12 章段 12.20 的开头部分讨论过这样的尾引用。本应用中使用的线性表类将做这样的调整。

**方法 `display`**。之前，我们选择含有迭代器的线性表实现，以便下面的 `display` 方法可以高效地显示索引中的行号。注意到，我们使用了字典迭代器，类似于前面示例中，在段 20.17 所给的 `display` 方法中用到的。不过这里，每个值是一个线性表，它有自己的迭代器，可用来遍历线性表中的行号。

```

public void display()
{
 Iterator<String> keyIterator = wordTable.getKeyIterator();
 Iterator<ListWithIteratorInterface<Integer>> valueIterator =
 wordTable.getValueIterator();

 while (keyIterator.hasNext())
 {
 // Display the word
 System.out.print(keyIterator.next() + " ");

 // Get line numbers and iterator
 ListWithIteratorInterface<Integer> lineList = valueIterator.next();
 Iterator<Integer> listIterator = lineList.getIterator();

 // Display line numbers
 while (listIterator.hasNext())
 {
 System.out.print(listIterator.next() + " ");
 } // end while
 System.out.println();
 } // end while
} // end display

```

20.21



学习问题 9 为类 `Concordance` 编写方法 `getLineNumbers`，它返回含有给定单词的行号线性表。

## Java 类库：接口 Map

**20.22** 标准包 `java.util` 中含有接口 `Map<K,V>`，它类似于我们的用于 ADT 字典的接口。下列方法头选自 `Map` 中的方法，类似于在本章中见过的一些。我们在不同于我们方法的地方做了标注。

```
public V put(K key, V value);
public V remove (Object key);
public V get(Object key);
public boolean containsKey(Object key);
public boolean containsValue(Object value);
public Set<K> keySet();
public Collection<V> values();
public boolean isEmpty();
public int size();
public void clear();
```

注意到方法名的不同。`Map` 使用方法名 `put`、`get`、`containsKey` 和 `size`，分别替代我们的名字 `add`、`getValue`、`contains` 和 `getSize`。`Map` 还有另外的方法 `containsValue`，它查看字典中是否含有给定的值。

不同于我们的方法 `getKeyIterator` 和 `getValueIterator`，分别返回用于字典的查找键和值的迭代器，`Map` 中规范说明了返回查找键集合的方法 `keySet` 和返回值集合的方法 `values`。`Java` 类库中含有接口 `Set` 和 `Collection`，每个接口都有方法 `iterator`，它返回相应 ADT 中值的迭代器。

实现 `Map` 接口的字典中不允许有重复的查找键。每个查找键必须对应于唯一的值。另外，`Map` 中的有些方法使用 `Object` 作为查找键的数据类型，而我们使用更一般的泛型数据类型 `K`。

## 本章小结

- ADT 字典中的每个项都含有两部分：查找键和对应于查找键的值。字典由其查找键识别它的项。
- 英语字典、电话号码簿、地址簿，及图书目录都是常见的字典示例。
- 可以将给定的查找键及其值组成的项添加到字典中。可以仅给出查找键，获取或删除该项。通过使用迭代器，可以遍历字典中所有的查找键或所有的值。
- 字典可以按查找键有序或无序来组织。查找键可以唯一也可以重复。
- 字典是否含有有序或无序的查找键，是影响其操作效率的实现细节。
- `Java` 类库中含有接口 `Map`，它类似于我们的 `DictionaryInterface`。

## 程序设计技巧

- 类 `Scanner` 能将字符串分隔为子串，或称为记号，它们由称为分隔符的符号分隔。默认情况下，分隔符是空格。可以将要分析的字符串传给 `Scanner` 的构造方法，或是将表示为 `java.io.File` 实例的一个文本文件传给它。
- `Scanner` 类中的下列方法能从任何字符串中提取记号。

```
public String next();
public boolean hasNext();
```

补充材料 1（在线）中从段 S1.81 开始详细讨论了 `Scanner`。

- 当使用 `Scanner` 对象来处理文本时，任何没有出现在所需记号中的字符都可以是分隔符。你可以使用一个特殊的记号来创建这些分隔符，并将它传给 `Scanner` 的方法 `useDelimiter`。更多细节请参看补充材料 1 (在线) 中的段 S1.82。

## 练习

- 字典区别于有序表的特点有哪些？
- 为段 20.9 描述的类 `TelephoneDirectory` 实现一个方法，给定人名及电话号码，将由此组成的项添加到电话簿中。如果项添加成功，则方法应该返回真。如果人名已在电话簿中，则方法应该替换其电话号码并返回假。
- 为段 20.9 的 `TelephoneDirectory` 类实现一个方法，显示每个人的名字及电话号码。
- 在段 20.7 的电话号码簿问题中，名字中的字母大小写影响名字在字典中的次序。你能采取什么处理步骤，让输入文件中的大小写的改变不影响它们的次序？
- 在段 20.7 的电话号码簿问题中，假定名字和电话号码的文本文件是按名排序的。
  - 对于字典的不同实现方式，这个文件的哪些方面将影响方法 `readFile` 的效率？
  - 文件按逆字典序排列会有关系吗？
- 逆电话簿 (reverse directory) 能查找对应于给定电话号码的名字。修改段 20.9 中的 `TelephoneDirectory` 类，让其具有这个功能。使用第二个字典当作逆电话簿。添加查询方法，并相应地修改 `readFile` 方法。
- 画出段 20.13 中概述的 `FrequencyCounter` 类的类图，它类似于段 20.8 的图 20-4 中的类图。
- 段 20.12 中的单词 - 频度问题，找到给定文本文件中出现的每个不同单词的频度。如果你想对每个频度列出相应的单词，描述你对类 `FrequencyCounter` 所做的修改。
- 对段 20.19 中概述的类 `Concordance`，重做练习 7。
- 在段 20.18 的索引问题中，如果单词在一行中出现多次，则索引中出现的行号也会多于 1 次。修改段 20.20 中所给的 `readFile` 方法，使得对应于一个给定单词的行号都是不同的。
- 设计一个保存不同药物副作用的 ADT。每种药物应该有对应于副作用的一个线性表。提供一个方法，返回给定药物的副作用。然后使用字典实现类 `DrugSideEffects`。
- 考虑查找给定日期电视播放节目的服务。一个文件中含有这些节目的信息。每个节目的数据分两行显示。第一行是电台名、频道、开播时间、结束时间、节目名及等级。这些项由波浪线 (~) 分隔，时间采用 24 小时表示（例如，下午 1 点是 13:00）。第二行主要描述这个节目。  
实现有下列方法头的方法

```
public void readFile(Scanner data)
```

将文件读入要被查找的字典中。决定哪些数据应该是查找键，哪些应该是对应的值。为这些查找键和值设计必要的类。

- 本章讨论的 ADT 字典假定有唯一的查找键。修改字典的规范说明，去掉这个限制。考虑下列每种可能性：
  - 方法 `add` 将其查找键已在字典中但尚没有值的一个项添加到字典中。方法 `remove` 删除含有给定查找键的所有项。方法 `getValue` 获取含有给定查找键的所有项。
  - 方法具有问题 a 中描述的行为，但使用第二个查找键，使得 `remove` 和 `getValue` 仅删除或获取一个项。

## 项目

- 定义实现程序清单 20-1 给出的 `DictionaryInterface` 的无序字典类 `OurDictionary`。在平

行的数据结构中维护查找键和对应的值，比如两个线性表或一个向量和一个线性表。

2. 定义实现程序清单 20-1 给出的 `DictionaryInterface` 的有序字典类 `OurSortedDictionary`。在平行的数据结构中维护查找键和对应的值，比如一个有序表和一个线性表。

在下列项目中当你需要使用字典时，使用项目 1 中要求你定义的 `OurDictionary` 类，或是项目 2 中要求你定义的 `OurSortedDictionary` 类。

3. 为简化段 20.7 中的电话号码簿问题，我们假定文本文件含有不同的名字。去掉这个假设，带第二个查找键及不带第二个查找键。(见练习 13。)
4. 发现著名的文学作品的作者是一个有趣的问题。在有争议的及知名的作者的作品之间进行比较。一种方法是在字母对的频度之间进行比较。共有  $26 \times 26$  个不同的字母对。并不是所有的字母对都出现在作品中。例如“qz”不太可能出现，而“th”则常常出现。设计一个程序，类似于段 20.12 到段 20.17 讨论的频度计数问题，统计给定文本中出现的所有字母对。
5. 假定我们想实现 ADT 集合。回忆第 1 章项目 1 中的定义，集合是对象的无序集合，其中不允许有重复值。集合应该支持的操作是

- 将给定对象添加到集合中
- 从集合中删除给定对象
- 查看集合中是否包含给定对象
- 从集合中清除所有对象
- 得到集合中对象的个数
- 为集合返回一个迭代器
- 返回包含两个集合中项的集合（合并）
- 返回由同时出现在两个集合中的项组成的集合（交）

在程序清单 1-5 所给的接口 `SetInterface` 中增加这些操作。然后定义实现 `SetInterface` 的类 `DictionarySet`，内部使用字典来实现这些操作。

6. 假定想帮助医生诊病。医生发现病人的症状，考虑可能与这些症状相关的疾病。设计并实现一个类 `PhysiciansHelper`，提供这些疾病线性表。

`PhysiciansHelper` 应该含有疾病和症状的字典。有一个方法应该将疾病及对应症状的文本文件读到字典中。文件中的每一行将含有疾病的名字，后面是冒号，然后是由逗号分隔的症状线性表。例如，一行可能如下

```
head cold: nasal stuffiness, sneezing, runny nose
```

`PhysiciansHelper` 应该维护当前病人的症状线性表。有一个方法应该将症状添加到这个线性表中，并返回对应于这些症状的疾病线性表。另一个方法应该从线性表中删除给定症状，还有一个方法应该清空病人症状线性表。

7. 写一个对弈 tic-tac-toe (井字棋——译者注) 游戏的程序。使用 9 值数组表示游戏盘。数组的每个位置含有 X 或是 O 或是空格。游戏盘不同的状态总数是  $3^9$ ，约为 20 000。对应于每种可能状态的是最佳走步。

生成所有可能的游戏盘，让它们作为字典中的查找键。对每一个查找键，让下一步最佳走步作为对应的值。一旦创建了这个字典，用它来决定 tic-tac-toe 游戏中计算机一方的走步。

8. 图解字典是图像集合，每幅图由描述字标识。使用从网上找到的免费图片创建外存文件，并由文件中的数据构成图解字典。设计并实现用户接口，提供查找和显示功能。
9. 使用下列修改重做第 10 章项目 17。我们知道用户自定义的标识符或符号，是程序中所有变量、常量、对象及方法的名字。编译程序不是像项目 17 那样将这些标识符添加到一个线性表中，而是建立一个符号表 (symbol table)。本项目中，符号表是一个字典。这个字典中的每个项都有一个标识符用作查找键，而对应的值是指向符号所表示的程序组件的引用。当编译程序遇到一个标识符时，

它检查另一个字典，看看标识符是不是保留字。如果不是，它再检查它是不是已经出现在符号表（字典）中。如果标识符没有出现在符号表中，则编译程序将标识符添加到表中。另一方面，如果标识符已经在字典中，则编译程序必须分析它遇到标识符的上下文。如果它在新的声明语句中，编译程序遇到了一个程序错误所以会发出一条出错信息。否则，它查阅字典中有关标识符的数据，然后继续编译程序。

设计并实现类 `ProgramSymbol`，这是其他符号类的基类。设计类 `SymbolTable`，维护给定的 Java 程序中由程序员定义的所有符号。

10. 设计并实现一个关于朋友及亲戚的数据库。为每个人保存的数据必须包含名字及至少一条其他的信息，比如生日。可以假定名字是唯一的。数据库应该能添加、删除、修改或查找数据。还应该能将数据保存在文件中以备将来使用。

定义表示一个人的类 `Person`，及表示数据库的另一个类。字典应该是数据库类的数据成员，`Person` 对象作为这个字典的查找键。写程序，测试并说明你的数据库。

你可以增加一个操作来强化这个项目，列出满足给定条件的所有人。例如，可以列出给定月份出生的所有人。还可以列出数据库中的所有人。

11. 考虑你至少可用两种方法组织的数据集合。例如，可以按姓名或身份证号排序的雇员，或是按书名或作者排序的书籍。注意，有关雇员或书籍的其他信息可以出现在数据库中，但不能用来组织这些项。查找键是字符串且是唯一的。所以，在刚才提到的例子中，身份证号必须是一个字符串，而不是一个整数，并且每位作者只允许有一本书。选择满足这些要求的数据集合，并创建一个文本文件。

程序行为。当程序开始运行时，它应该读入文本文件。然后应该提供一些典型的数据库管理操作，所有这些都通过你设计的界面由使用者控制。例如，应该能添加一个项、删除一个项、显示（即获取）一个项并按查找键次序显示所有的项。应该能使用两个查找键中的任何一个来指定要被删除或显示的项。

实现说明。数据库中的项应该是含有两个查找键及其他数据的对象，所有这些都出现在文本文件中。所以你必须设计并实现这些对象的一个类。

虽然你的程序可以从这些对象中创建两个字典——其中一个按一个查找键（比如雇员姓名）组织而另一个按另一个查找键组织（比如身份证号）——但这个方法会浪费大量内存，因为两个字典中所有的数据都是重复的。这还可能导致数据的一致性，如果程序员错误地仅更新了一个字典中的数据而没有更新另一个字典时。

有一种更好的方法修改 ADT 字典，为的是可以根据两个查找键提供操作。例如，你想根据姓名或根据身份证号进行删除。用来实现字典的底层数据结构可以是两个其他的字典，或是你自己设计的一个字典，这样你能以两种方式组织数据：比如按姓名及按身份证号。为避免重复数据，将数据保存在线性表中，并让每个字典项含有数据在线性表中的位置而不是数据本身。

你的程序可以特定于数据库类型（雇员、书籍等），或是更一般的类型。例如，用户界面显示的查找键描述可以放在文本文件中。

# 字典的实现

**先修章节：**第 3 章、第 4 章、第 10 章、第 11 章、第 12 章、Java 插曲 4、第 19 章、第 20 章

## 目标

学习完本章后，应该能够

- 使用数组或是结点链表实现 ADT 字典

本章提出的 ADT 字典的实现，用到了实现 ADT 线性表时用过的技术。我们将把字典的项保存在数组或是结点链表中。为此，考虑具有唯一查找键的有序字典和无序字典。后面的章节将提出 ADT 字典更复杂的实现方式。

## 基于数组的实现

21.1

第 2 章段 2.35 中介绍的变长数组的能力，意味着数组可以按照字典中项的需要提供存储。记住，每个项包含两部分——一个查找键和一个值。可以将这两部分封装到一个对象中。如图 21-1a 所示。使用这个方法，定义类 Entry 来表示项。第二种，不是太让人喜欢的方法是使用两个数组，如图 21-1b 所示。一个数组用来表示查找键，第二个平行数组 (parallel array) 用来表示对应的值。我们将讨论第一种方法，而将第二种方法的研究留作练习。到那时，你会看到平行数组不易管理。

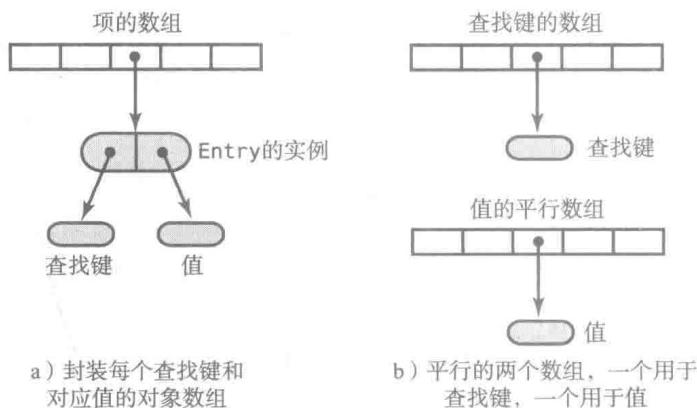


图 21-1 使用数组表示字典中项的两种可能方法



**学习问题 1** 图 21-1 显示了表示基于数组实现字典的两种方法。比较一下，两种表示所需的内存各是多少？

## 基于无序数组的字典

21.2

开始实现。我们的实现使用如图 21-1a 所示的一个数组来表示字典。字典，即数组中的

每个项，都是我们必须定义的类 Entry 的一个实例。可以让这个类是公有的，成为包的一部分，也可以是字典类的内部私有类。我们选择后一种方式，定义私有类 Entry，如程序清单 21-1 所示。

外部类 ArrayDictionary 的最前面是数据域和由类型参数 K 和 V 表示的构造方法。这些参数分别表示查找键及对应值的数据类型。

### 程序清单 21-1 类 ArrayDictionary 和其私有内部类 Entry

```

1 import java.util.Arrays;
2 import java.util.Iterator;
3 import java.util.NoSuchElementException;
4 /**
5 * A class that implements the ADT dictionary by using a resizable array.
6 * The dictionary is unsorted and has distinct search keys.
7 * Search keys and associated values are not null.
8 */
9 public class ArrayDictionary<K, V> implements DictionaryInterface<K, V>
10 {
11 private Entry<K, V>[] dictionary; // Array of unsorted entries
12 private int numberOfEntries;
13 private boolean integrityOK = false;
14 private final static int DEFAULT_CAPACITY = 25;
15 private static final int MAX_CAPACITY = 10000;
16
17 public ArrayDictionary()
18 {
19 this(DEFAULT_CAPACITY); // Call next constructor
20 } // end default constructor
21
22 public ArrayDictionary(int initialCapacity)
23 {
24 checkCapacity(initialCapacity);
25 // The cast is safe because the new array contains null entries
26 @SuppressWarnings("unchecked")
27 Entry<K, V>[] tempDictionary = (Entry<K, V>[])new Entry[initialCapacity];
28 dictionary = tempDictionary;
29 numberOfEntries = 0;
30 integrityOK = true;
31 } // end constructor
32
33 <Implementation of methods in DictionaryInterface. >
34 . . .
35
36 private class Entry<K, V>
37 {
38 private K key;
39 private V value;
40
41 private Entry(K searchKey, V dataValue)
42 {
43 key = searchKey;
44 value = dataValue;
45 } // end constructor
46
47 private K getKey()
48 {
49 return key;
50 } // end getKey
51
52 private V getValue()
53 {
54 return value;

```

```

55 } // end getValue
56
57 private void setValue(V newValue)
58 {
59 value = newValue;
60 } // end setValue
61 }
62 } // end ArrayDictionary

```

注意到，内部类 Entry 没有方法 setKey 去设置或修改查找键。即使 setValue 在 add 的实现中很有用，但你也从不需要修改查找键。没有 setKey 方法，默认构造方法就没有用了，因此也就没有定义。

### 注：编译程序警告

程序清单 21-1 中所示的 ArrayDictionary 的构造方法，使用表达式语句 new Entry [initialCapacity] 为数组 dictionary 分配内存。编译程序发现数组元素的类型是 Entry。当构造方法将这个数组赋值给其元素是 Entry<K,V> 类型的数组时，编译程序提示一个未经检查的转换。若试图将新数组转型为 Entry<K,V>[] 时也会得到类似的警告。尽管有编译程序的警告，但这两种情况都没有错。所以，我们禁止这个警告，与过去构造方法将 Object 实例转型为泛型时的处理一样。

#### 21.3

**一些私有方法。**基于数组实现 ADT 的一个问题是数组大小的有限性。为避免字典满了，我们根据需要将数组的大小扩大一倍，如同我们在前面几章的做法一样。然后会使用一个私有方法，这也和之前的做法一样。它的规范说明如下所示。

```
// Doubles the size of the array of entries if it is full.
private void ensureCapacity()
```

添加、删除或是获取一个项时，因为查找键是无序的，故需要进行顺序查找。顺序查找必须查看数组中的所有项，以推断出一个项目前在不在字典中。将这个查找方法实现为如下的私有方法，会简化这三个字典操作的定义。

```
// Returns the array index of the entry that contains key, or
// returns numberofEntries if no such entry exists.
private int locateIndex(K key)
```

最后，为加强代码的安全性，我们还定义了私有方法 checkCapacity 和 checkIntegrity，如同之前基于数组实现中所做的一样。

#### 21.4

**添加一个项。**基于数组实现的另一个潜在问题是，数组项的移动是经常发生的。但当字典的查找键无序时，添加或删除项时无须移动其他的项。所以，当添加新的键 - 值项时，可以将其插入在数组最后一项的后面，如图 21-2 所示。这种情形下，add 返回 null。但，如果查找键已经在字典中，则我们用新的值替换相应的值，并返回原来的值。下列算法实现了这些步骤。

```
Algorithm add(key, value)
// Adds a new key-value entry to the dictionary and returns null. If key already exists
// in the dictionary, returns the corresponding value and replaces it with value.
// key and value are not null.

result = null
在字典中查找含有key的项
```

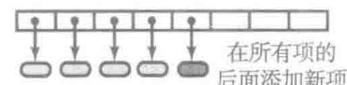


图 21-2 将新项添加到基于无序数组的字典中

```

if (找到含有key的项)
{
 result = 当前对应于 key 的值
 用 value 替换 key 对应的值
}
else // Insert new entry

if (数组已满)
 增倍数组大小
在数组最后一项的后面插入含有key和value的新项
字典大小加1
}
return result

```



### 设计决策：应该如何防止查找键和值是 null？

在第 20 章当为字典设计接口时，我们决定禁止其查找键或值是 null 的项。我们还在 add 方法的前一个伪代码的注释中注明了这个决策。这个方法是唯一一个将键 - 值对作为项添加到字典中的，所以在这里避免 null 的键或值是很重要的。但因为 add 是公有方法，所以我们需要做的不仅仅是陈述一个希望客户会遵守的前置条件。

因为方法 add 将创建一个新的 Entry 对象，那么应该让 Entry 的构造方法来检查 null 键和值吗？有以下充分的理由，不让 Entry 来担这个职责：

- add 要做的第一件事，是在字典中查找包含给定查找键的项。它必须要这样做，以避免多个项有相同的键。只有在查找完成，并且断定查找键不在字典中后，add 才创建一个新的 Entry 对象。如果键或其对应的值已经是 null 了，则在 Entry 的构造方法不接受这个项之前，我们可能已经查找了上百万的字典项。在构造方法检查之前检查 null 应该是很好的。
- Entry 是我们可能用在其他 ADT 实现中的类，或者用在允许有 null 查找键或值的字典中。因此，它应该是通用的，且可以是包的一部分，而不是一个内部类。测试查找键和值的合法性应该在 Entry 的外面进行，因为测试的准则应该由 Entry 的客户设计决定。

我们决定，方法 add 不会向字典中添加 null 的查找键或值。

add 方法。add 方法的下列实现调用了私有方法 checkIntegrity，还调用了段 21.3 说明的私有方法 locateIndex 和 ensureCapacity。21.5

```

public V add(K key, V value)
{
 checkIntegrity();
 if ((key == null) || (value == null))
 throw new IllegalArgumentException();
 else
 {
 V result = null;
 int keyIndex = locateIndex(key); // key is not null
 if (keyIndex < numberofEntries)
 {
 // Key found; return and replace entry's value
 result = dictionary[keyIndex].getValue(); // Get old value
 dictionary[keyIndex].setValue(value); // Replace value
 }
 else // Key not found; add new entry to dictionary
 {

```

```

 // Add at end of array
 dictionary[numberOfEntries] = new Entry<>(key, value);
 numberOfEntries++;
 ensureCapacity(); // Ensure enough room for next add
} // end if
return result;
} // end if
} // end add

```

为查找无序数组，`locateIndex` 的定义如下。

```

// Precondition: key is not null.
private int locateIndex(K key)
{ // Sequential search
 int index = 0;
 while ((index < numberOfEntries) &&
 !key.equals(dictionary[index].getKey()))
 index++;
 return index;
} // end locateIndex

```

这个方法有一个前置条件，可以保证 `while` 语句中 `key` 与字典中的查找键进行比较时不会抛出异常。

#### 21.6

**删除一项。**要从基于无序数组的字典中删除一项，首先要找到这项，然后用字典的最后一项来替换它，如图 21-3 所示。所以，我们不需要移动其他的项，就可以填充数组中的“空位”。因为字典的大小减 1，故原来指向当前项的引用将被忽略。但是为了安全考虑，我们将那个引用置为 `null`。

下列算法描述删除操作。

```

Algorithm remove (key)
// Removes an entry from the dictionary, given its search key, and returns its value.
// If no such entry exists in the dictionary, returns null.
result = null
在数组中查找含有key的项
if (在数组中找到含有key的项)
{
 result = 当前对应于key的值
 使用数组中最后一项替代这个项
 将含最后一项的数组元素置为null
 字典大小减1
}
// Else result is null
return result

```

#### 21.7

**其他的方法。**将字典实现中的其他方法留作练习，因为一旦你已经学到了现在，那么，那些工作并不难实现。注意，字典项的迭代或遍历，就是简单地在数组内从一个位置移动到另一个位置。因为查找键无序，所以迭代次序并不是确定的。无论什么次序，易于实现就是好的。一般地，从数组的第一项开始，顺序移动过其余的项。

#### 21.8

**效率。**对于这个实现，各操作最差情况下的效率如下所示。

添加  $O(n)$

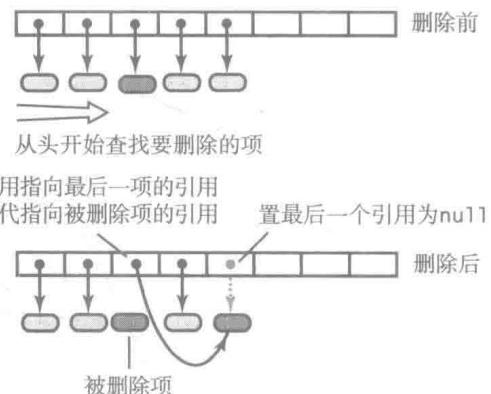


图 21-3 从基于无序数组的字典中删除一项

删除  $O(n)$

获取  $O(n)$

遍历  $O(n)$

虽然添加操作是将项添加在数组 `dictionary` 最后一项之后而不需要移动任何数据，但查找是必需的，为的是防止字典中有重复查找键出现，这使得整个操作是  $O(n)$  的。删除和获取也用到了类似的对数组的查找，所以也是  $O(n)$  的。最后，遍历数组是  $O(n)$  操作。

要知道，如果字典项填满了数组，则必须分配新的更大的数组，并将项从原数组中复制到新数组中。这个需求增加了基于数组实现的开销，而这些因素在前面的分析中并没有考虑进去。在 Java 中，数组项是指向对象的引用，所以数组的复制是快速的。理想情况下，你要选择一个足够用的数组，而不是选一个因太大而浪费空间的数组。

## 基于有序数组的字典

无序字典的某些实现，如段 21.2 中所示的，不依赖于字典项的次序，所以可用在有序字典中。不过，现在查找键必须属于实现了接口 `Comparable` 的类，所以我们可以排序它。有序字典的实现框架列在程序清单 21-2 中。最初在 Java 插曲 5 的段 J5.10 中介绍过的符号 `K extends Comparable<? super K>`，定义了泛型 `K`。这允许我们将类型 `K` 的对象与类型 `K` 或是 `K` 的父类的任何对象进行比较。

21.9

**程序清单 21-2** 类 `SortedDictionary` 的框架

```

1 import java.util.Arrays;
2 import java.util.Iterator;
3 import java.util.NoSuchElementException;
4 /**
5 * A class that implements the ADT dictionary by using a resizable array.
6 * The dictionary is sorted and has distinct search keys. Search keys and
7 * associated values are not null.
8 */
9 public class SortedDictionary<K extends Comparable<? super K>, V>
10 implements DictionaryInterface<K, V>
11
12 < Data fields as shown in Listing 21-1 of Segment 21.2. >
13
14 . .
15
16 public V add(K key, V value)
17 {
18 . . . < See Segment 21.11. >
19 } // end add
20
21 < Implementations of other methods in DictionaryInterface. >
22
23 . .
24
25 < The private class Entry, as shown in Listing 21-1. >
} // end SortedDictionary

```

**添加一项。**当字典的键 – 值项按查找键有序时，新项的添加需要先查找项的数组，看看新项应处的位置。为新项判定正确的位置后，还必须为它在数组中腾出空间。为此，将数组中随后的项后移一个位置，从最后一项开始，如图 21-4 所示。然后将新项插入数组中，这样它处于按查找键有序的正确位置。

21.10

下面的添加项的算法类似于段21.4中给出的用于无序字典的方法。

```
Algorithm add(key, value)
// Adds a new key-value entry to the dictionary and returns null. If key already exists
// in the dictionary, returns the corresponding value and replaces it with value.
如果key或value是null, 则抛出一个异常
result = null
查找数组, 直到找到含有key的项, 或是找到它应该在的位置
if (在数组中找到含有key的项)
{
 result = key当前对应的值
 使用value替换key对应的值
}
else // Insert new entry
{
 在数组中根据前面的查找指示的下标位置为新项腾出空间
 将含有key和value的新项插入数组中腾空的位置
 字典的大小加1
 if (数组满了)
 倍增数组
}
return result
```

### 学习问题2 叙述前一个算法与段21.4中给出的用于无序字典的算法的不同之处。

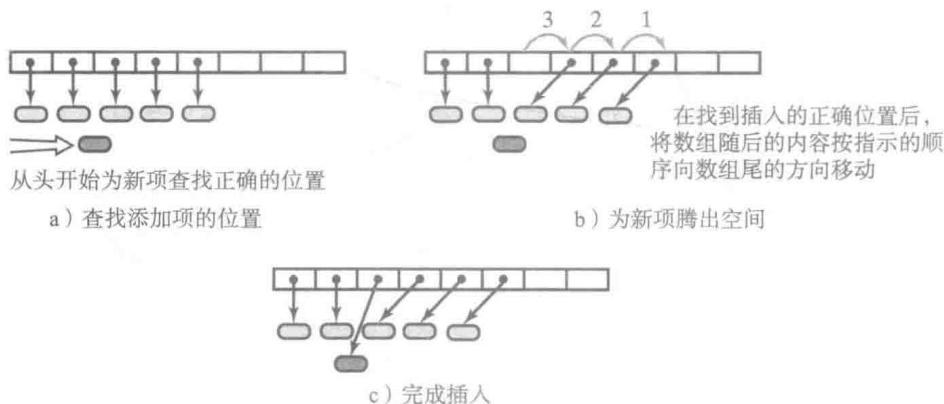


图21-4 将项添加到基于有序数组的字典中

**21.11 方法add。**可以使用段21.3中描述的私有方法来实现这个算法，但需要用到`locateIndex`的另一种实现。当字典无序时，`locateIndex`只是检查字典中是否含有给定的查找键。但这里，`locateIndex`还必须找到插人在数组的哪个位置。所以我们修改方法的规范说明，如下所示。

```
// Returns the index of either the entry that contains key or
// the location that should contain key, if no such entry exists.
private int locateIndex(K key)
```

下面这个额外的方法也有助于类的实现：

```
// Makes room for a new entry at a given index by shifting
// array entries towards the end of the array.
private void makeRoom(int keyIndex)
```

使用这些方法，可以实现`add`方法，如下所示。

```

public V add(K key, V value)
{
 checkIntegrity();
 if ((key == null) || (value == null))
 throw new IllegalArgumentException();
 else
 {
 V result = null;
 int keyIndex = locateIndex(key);
 if ((keyIndex < numberofEntries) &&
 key.equals(dictionary[keyIndex].getKey()))
 {
 // Key found; return and replace entry's value
 result = dictionary[keyIndex].getValue(); // Get old value
 dictionary[keyIndex].setValue(value); // Replace value
 }
 else // Key not found; add new entry to dictionary
 {
 makeRoom(keyIndex);
 dictionary[keyIndex] = new Entry<>(key, value);
 numberofEntries++;
 ensureCapacity(); // Ensure enough room for next add
 } // end if
 return result;
 } // end if
} // end add
}

```

这个方法和段 21.5 中所给的用于无序字典的方法的不同之处，已经标注出来。

**方法 locateIndex。**因为数组是有序的，所以比起在无序数组中的查找，`locateIndex`通常可以花更少的时间。回顾第 19 章段 19.8，当项不在有序数组中时，顺序查找不需要查找整个数组就可以判定。使用这个方法，可以定义私有的 `locateIndex` 方法，如下所示。

```

private int locateIndex(K key)
{
 // Search until you either find an entry containing key or
 // pass the point where it should be
 int index = 0;
 while ((index < numberofEntries) &&
 key.compareTo(dictionary[index].getKey()) > 0)
 index++;

 return index;
} // end locateIndex

```

这个方法和段 21.5 中所给的用于无序字典的方法的不同之处，已经标注出来。

 **学习问题 3** 一般来讲，二分查找比刚刚给出的修改后的顺序查找更快——特别是当字典很大时。使用二分查找为有序字典实现私有方法 `locateIndex`。

**删除一项。**从基于有序数组的字典中删除一项，要先调用 `locateIndex` 方法找到这个项，这个方法我们在前面 `add` 方法中使用过。因为项是有序的，所以我们必须维护这个次序。故被删除项后面的任何项都必须前移到数组中更低的一个位置中。图 21-5 说明了这两步。

下面的算法描述了删除操作。

```

Algorithm remove (key)
// Removes an entry from the dictionary, given its search key, and returns its value.
// If no such entry exists in the dictionary, returns null.

result = null
在数组中查找含有key的项

```

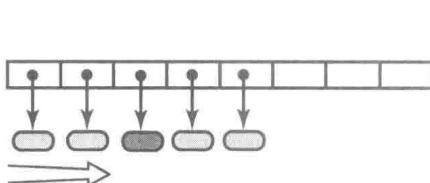
21.12

21.13

```

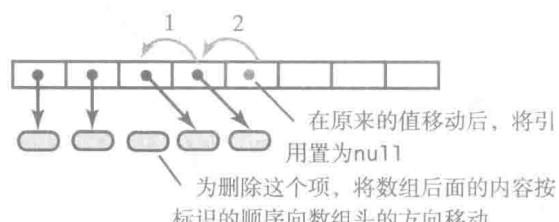
if (在数组中找到含有key的项)
{
 result = key当前对应的值
 将找到的项后面的所有项移向数组中前一个较低位置
 将含有最后一项的数组元素置为null
 字典大小减1
}
return result

```



从头开始查找被删除项

a) 查找要删除的项



在原来的值移动后，将引用置为null  
为删除这个项，将数组后面的内容按标识的顺序向数组头的方向移动

b) 向被删除的元素方向移动项

图 21-5 从基于有序数组的字典中删除一项

将这个算法的实现留作练习。定义下面的私有方法是有帮助的。

```

// Removes an entry at a given index by shifting array
// entries toward the entry to be removed.
private void removeArrayEntry(int keyIndex)

```

21.14

**其余的方法。**对于给定查找键获取已存在项的值的方法 `getValue`，其核心是方法 `locateIndex`，如前所述。因为数组项有序，故 `locateIndex` 可以使用二分查找，如学习问题 3 说明的。

字典项的迭代或遍历，从数组的第一项开始，顺序移过所有的项。这部分的实现可以与无序字典中的方法一样。但这里，因为数组是有序的，故迭代将按查找键有序的方式遍历字典。

我们将这个方法的实现留作练习。

21.15

**效率。**在基于有序数组的字典实现中，当 `locateIndex` 使用二分查找时，字典操作的最差情况效率如下。

添加  $O(n)$

删除  $O(n)$

获取  $O(\log n)$

遍历  $O(n)$

这个实现适用于创建字典后多次取值的应用。第 12 章设计决策中的观点在这里重复一遍：



**程序设计技巧：**当选择 ADT 的实现时，应该考虑应用所需要的的操作。如果频繁用到某一个 ADT 操作，就应该让它的实现更有效。相反，如果很少用到一种操作，则可以使用低效率实现那种操作的类。



**程序设计技巧：**在类的实现中要包含说明这个方法效率的注释。



**学习问题 4** 当基于有序数组实现字典时使用二分查找，它的获取操作是  $O(\log n)$  的。而 `add` 和 `remove` 使用了同样的查找机制，为什么它们不是  $O(\log n)$  的？

## 链式实现

本章讨论的 ADT 字典的最后一种实现，是将字典项保存在结点链表中。如第 3 章提出的，链表可以根据项的需要量而提供存储。可以将项的两部分封装为一个对象，如图 21-6a 所示，与数组中使用的一样。如果选择这样做，则字典类可以使用段 3.25 中的 Node 类和程序清单 21-1 中的 Entry 类。

另一种选择是不使用类 Entry。可以使用两个链表，如图 21-6b 所示，但更简单的方法是修改结点的定义，让其包含项的两个部分，如图 21-6c 所示。定义在字典类内的私有内部类 Node 应该含有数据域。

```
private K key;
private V value;
private Node next;
```

泛型 K 和 V 由外部类来定义。除了构造方法外，类 Node 应该含有方法 getKey、getValue、setValue、getNextNode 和 setNextNode。因为不需要修改查找键，且事实上也不能破坏有序字典的次序，所以没有提供 setKey 方法。

## 无序链式字典

因为无序字典中的项没有特定的次序，所以可以以最高效的方式添加新项，而不需要关心新项在字典中的位置。当项保存在如图 21-6c 所示的那种链表中时，最快的添加是在链表表头处执行的，如图 21-7 所示。(如果类还维护着指向链表表尾结点的尾引用，则在最后结点的后面添加也同样快。) 而这种情形下，添加是  $O(1)$  的，防止重复查找键将需要从链表表头开始的顺序查找。与数组中的情形一样，你必须查看链表中的所有查找键，以了解某个项不在链表中。

删除或获取一项，用到同样的查找。遍历查找键或是值，将涉及整个的链表。所以对它们的实现，最差情况下操作的效率如下所示。

添加  $O(n)$

删除  $O(n)$

获取  $O(n)$

遍历  $O(n)$

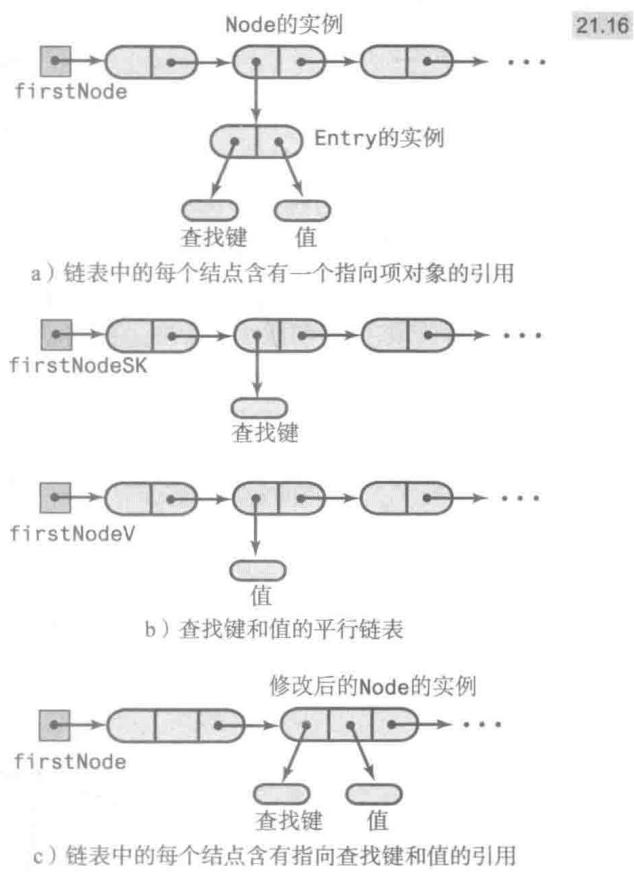


图 21-6 使用结点链表表示字典中的项的 3 种可能方法

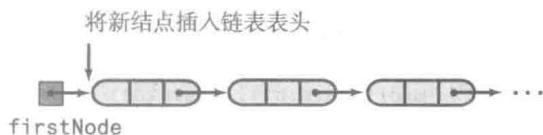


图 21-7 添加到无序链式字典中



**学习问题 5** 要从基于无序数组的字典中删除一项，可以用数组的最后一项来替换被删除的项（见 21.1.1 节）。应该使用同样的策略从无序链式字典中删除一项吗？请解释之。

## 有序链式字典

**21.18** 添加一项。当链表中的结点按查找键有序时，将新项添加到字典中需要从链表表头开始顺序查找链表，以确定新结点的正确位置。因为查找键有序，所以一旦越过应该包含它的结点时，就可以马上判定要找的查找键不存在。即你不必像查找键无序时那样查看整个链表。第 19 章的段 19.8 和段 19.22 中，描述了修改后的这个顺序查找。

下列算法将一个新项添加到有序链式字典中。

```

Algorithm add(key, value)
// Adds a new key-value entry to the dictionary and returns null. If key already exists
// in the dictionary, returns the corresponding value and replaces that value with value.
如果key或者value为null，则抛出一个异常
result = null
查找链表，直到找到含有key的结点，或是越过了这个值应该在的位置
if (在链表中找到含有key的结点)
{
 result = key当前对应的值
 用value替换key对应的值
}
else
{
 分配一个新结点，包含key和value
 if (链表为空，或是新项位于链表表首)
 将新结点添加在链表表首
 else
 将新结点插入在查找时检查到的最后结点之前
 字典大小加1
}
return result

```

**21.19** 程序清单 21-3 是类 SortedLinkedListDictionary 的开始部分及 add 方法的实现。方法 remove 和 getValue 的实现类似于 add 的实现，但更简单一些。将它们留作练习。

### 程序清单 21-3 SortedLinkedListDictionary 类

```

1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3 /**
4 A class that implements the ADT dictionary by using a chain of linked nodes.
5 The dictionary is sorted and has distinct search keys.
6 Search keys and associated values are not null.
7 */
8 public class SortedLinkedListDictionary<K extends Comparable<? super K>, V>
9 implements DictionaryInterface<K, V>
10 {
11 private Node firstNode; // Reference to first node of chain
12 private int numberofEntries;
13
14 public SortedLinkedListDictionary()
15 {
16 initializeDataFields();
17 } // end default constructor
18

```

```

19 public V add(K key, V value)
20 {
21 V result = null;
22 if ((key == null) || (value == null))
23 throw new IllegalArgumentException("Cannot add null to a dictionary.");
24 else
25 {
26 // Search chain until you either find a node containing key
27 // or locate where it should be
28 Node currentNode = firstNode;
29 Node nodeBefore = null;
30 while ((currentNode != null) && key.compareTo(currentNode.getKey()) > 0)
31 {
32 nodeBefore = currentNode;
33 currentNode = currentNode.getNextNode();
34 } // end while
35
36 if ((currentNode != null) && key.equals(currentNode.getKey()))
37 {
38 result = currentNode.getValue(); // Get old value
39 currentNode.setValue(value); // Replace value
40 }
41 else
42 {
43 Node newNode = new Node(key, value); // Create new node
44 if (nodeBefore == null)
45 { // Add at beginning (includes empty chain)
46 newNode.setNextNode(firstNode);
47 firstNode = newNode;
48 }
49 else // Add elsewhere in non-empty chain
50 {
51 newNode.setNextNode(currentNode); // currentNode is after new node
52 nodeBefore.setNextNode(newNode); // nodeBefore is before new node
53 } // end if
54
55 numberOfEntries++; // Increase length for both cases
56 } // end if
57 } // end if
58 return result;
59 } // end add
60
61 < Implementations of the other methods in DictionaryInterface. >
62 . . .
63
64 < Private classes KeyIterator and ValueIterator (see Segment 21.20). >
65 . . .
66
67 < The private class Node. >
68 . . .
69
70 } // end SortedLinkedListDictionary

```

**迭代器。**迭代器为客户提供一种遍历字典中查找键和其对应值的简单方法。这里，**21.20** 公有方法 getKeyIterator 和 getValueIterator 的实现，与在其他字典中的实现是一样的。但私有内部类 KeyIterator 和 ValueIterator 的实现是不同的。每个类都有数据域 nextNode，它标记遍历过程中链表中的迭代位置。这两个类都很像第 13 章段 13.9 到段 13.13 中出现的内部类 IteratorForLinkedList。将它们的定义留作练习。

**效率。**与添加一个项一样，删除或获取一项时需要对链表进行顺序查找。**21.21** 有序链表的遍历与无序链表的遍历是一样的。所以最差情况下，有序链式实现的字典操作的效率如下所示。

添加  $O(n)$

删除  $O(n)$

获取  $O(n)$

遍历  $O(n)$

添加或删除项是  $O(n)$  操作，不管是使用数组还是使用链表来实现字典。但要知道，数组需要你移动其中的项，而链式实现不需要。另外，处理链式实现时不需要很好的估算字典的最大容量。当使用的数组太小时，可以通过将项拷贝到新的更大数组来扩展它，但这会浪费时间。如果使用的数组大于所需的量，会浪费空间。这两种情形在链式实现中都不会发生。

## 本章小结

- 可以使用数组或是结点链表实现字典。链式实现不需要很好的估算字典的最大容量。当使用的数组太小时，必须将项拷贝到新的更大数组中。如果使用的数组大于所需的量，会浪费空间。这两种情形在链式实现中都不会发生。
- 基于数组和链式实现的字典操作的最差情况效率如下所示。

|    | 基于数组的实现 |             | 链式实现   |        |
|----|---------|-------------|--------|--------|
|    | 无序      | 有序          | 无序     | 有序     |
| 添加 | $O(n)$  | $O(n)$      | $O(n)$ | $O(n)$ |
| 删除 | $O(n)$  | $O(n)$      | $O(n)$ | $O(n)$ |
| 获取 | $O(n)$  | $O(\log n)$ | $O(n)$ | $O(n)$ |
| 遍历 | $O(n)$  | $O(n)$      | $O(n)$ | $O(n)$ |

- 对于有序或无序的字典，添加或删除项是  $O(n)$  的操作，不论是使用数组还是链表来实现它。但是要知道，数组需要移动它的项，而链表不需要。
- 使用有序数组实现字典时可有高效的获取操作，因为你可以使用二分查找。
- 为实现方法 `getKeyIterator` 或是 `getValueIterator`，要为字典类定义一个私有内部类。这个私有类应该使用接口 `java.util.Iterator`。

## 程序设计技巧

- 当选择 ADT 的实现时，应该考虑应用需要的操作。如果频繁用到某一个 ADT 操作，就应该让它的实现更有效。相反，如果很少用到一种操作，则可以使用低效率实现那种操作的类。
- 在类的实现中包含说明这个方法效率的注释。

## 练习

- 根据图 21-1b 所示的数据结构，开始实现基于数组的 ADT 字典。声明数据域、定义构造方法，并为无序数据定义方法 `add`。使用运行期间可变长的数组。
- 根据图 21-6a 和图 21-6b 所示的两个数据结构，开始实现两种链式 ADT 字典。声明数据域、定义构造方法，并为无序数据定义方法 `add`。
- 对于程序清单 21-3 概述的链式实现的有序字典，编写迭代实现方法 `remove` 和 `getValue` 的代码。
- 重做前一个练习，这次要编写递归实现方法 `add`、`remove` 和 `getValue` 的代码。

5. 段 21.20 描述了内部类 `KeyIterator`。这个类的实例是一个迭代器，能遍历字典中的查找键。用类似的方式，内部类 `ValueIterator` 提供遍历字典中值的方法。为类 `SortedLinkedDictionary` 给出这两个内部类的定义。
6. 为 ADT 字典定义一个迭代器，返回含有查找键和值的项。描述这些项的类。实现返回这样的迭代器的方法 `getEntryIterator`。
7. 考虑 ADT 字典的附加操作，求给定的两个字典的并和交。每个操作都返回一个新字典。并应该将两个字典中的项合并到第三个字典中。交应该将同时出现在两个字典中的项放到一个字典中。  
在每个给定字典中，查找键不能重复。但是，一个字典中的项可以与第二个字典中的项有相同的查找键。规划并讨论这种情形下这两个操作的规范说明。
8. 对于基于无序数组实现的字典，实现练习 7 描述的并和交。
9. 对于基于有序数组实现的字典，重做练习 8。
10. 对于有序链式字典，重做练习 8。

## 项目

1. 实现基于无序数组的字典。运行期间允许数组按需扩展。
2. 重做前一个项目，但查找键要有序排列。
3. 使用 `Vector` 或是 `ArrayList` 的实例来保存字典项，实现无序字典。可以使用一个或两个向量或线性表，很像是图 21-1a 和图 21-6b 所示的一个或两个数组。因为 `Vector` 或是 `ArrayList` 的底层都是基于数组实现的，所以，不论是使用数组、线性表还是向量，用于字典操作的算法和它们的效率本质上是相同的。  
注，`Vector` 类和 `ArrayList` 类分别在第 6 章段 6.14 和第 10 章段 10.20 中介绍。
4. 重做前一个项目，但将查找键有序排列。当在字典中查找一个键时，使用二分查找替代顺序查找。
5. 使用结点链表实现无序字典。
6. 使用结点链表实现有序字典。
7. 本章，ADT 字典有不同的查找键。实现一个去掉这个限制的字典。选择第 20 章练习 13 所给的一种可能的实现方式。
8. 第 20 章段 20.7 开始讨论电话号码簿。使用你在前一个项目中实现的字典，修改电话号码簿，让它允许有重复的名字。
9. 图 21-1b 说明了如何使用平行数组来表示字典中的项。使用这种方法实现 ADT 字典。
10. 将段 21.2 的程序清单 21-1 中给出的 `Entry` 类，修改为实现了 `Comparable` 接口的公有类。比较两个 `Entry` 对象即是比较它们的查找键。使用这个类及 ADT 有序表的实现，编写有序字典的实现。这些类，包括 `Entry` 类，应该属于同一个包。
11. 重做前一个项目，但使用 `Entry` 对象的数组来替代有序表。
12. 重做项目 10，使用结点链表替代有序表，链表中每个结点含有指向 `Entry` 实例的引用。
13. 实现接口 `DictionaryInterface<String, String>`，创建词汇表类。词汇表是特定术语及其对应定义的字典。将词汇表表示为 26 个有序表的集合，每个字母对应一个表。词汇表中的每个项——包含一个术语和它的定义——保存在对应于术语首字母的有序表中。使用要作为词汇表的术语及定义组成的文本文件，认真测试你的类。
14. 实现第 20 章项目 9 中的类 `SymbolTable`。哪种字典的实现方式最适合这个字典的使用情形？

# 散列简介

先修章节：第 20 章、第 21 章

## 目标

学习完本章后，应该能够

- 描述散列的基本思想
- 描述散列表、散列函数和完美散列函数的目的
- 解释为什么应该为作为查找键使用的对象重写 `hashCode` 方法
- 描述散列函数如何将散列码压缩为散列表的地址
- 描述冲突并解释它们为什么会发生
- 描述解决冲突的开放地址法
- 描述开放地址机制中的线性探查、二次探查及双散列
- 描述开放地址解决冲突时字典操作 `getValue`、`add` 和 `remove` 的算法
- 描述解决冲突的拉链法
- 描述拉链法解决冲突时字典操作 `getValue`、`add` 和 `remove` 的算法
- 描述聚集及它引发的问题

因为查找数据库是计算机中如此普通的一种应用，故而字典是一种重要的抽象数据类型。第 21 章讨论的实现，对于某些应用还是不错的，但对其他一些应用就不能胜任了。例如，如果查找数据是关键操作，即使是  $O(\log n)$  的查找都嫌太慢。应急电话（911）系统就是这样一种情形。如果你从固定电话呼叫 911，那么你的电话号码就是在街道地址字典中进行查找的关键字。很显然，这个查找要能立即定位到你的地址！

本章介绍称为散列的技术，理想情况下，它能得到  $O(1)$  的查找时间。当查找是首要任务时，散列可以是实现字典时的极佳选择。下一章还将继续完善这个话题的讨论。

一方面散列可能非常好，但另一方面，它并不总是合适的。例如，散列不能提供查找键的有序遍历。本书后面还将讨论 ADT 字典的其他实现，这些确实能提供查找键的有序遍历。

## 什么是散列

22.1

物应各有其所；亦应各在其所。早上你有没有花时间找钥匙？或是你确切地知道它们在哪儿？我们有些人常常花很多时间按次序找自己未分类的东西。另外一些人将东西放在确定的地方而且知道到哪儿去找。

数组可以为字典项提供地方。诚然，数组有它自身的弱点，但如果知道了下标，就可以直接访问数组中的任何项。不需要涉及数组中的其他项。散列（hashing，或称哈希）是仅利用项的查找键，无需查找就可以确定其下标的一项技术。数组本身称为散列表（hash table）。

散列函数（hash function）根据查找键得到元素在散列表中的整数下标。这个数组元素是你应该保存或是查找对应于查找键的值的地方。例如，911 应急系统可以将你的电话号码转换为适当的整数  $i$ ，而数组元素  $a[i]$  中保存指向你街道地址的引用。我们称，电话号

码——即查找键——映射 (map) 或散列 (hash) 到下标  $i$ 。这个下标称为散列索引 (hash index)。有时我们称查找键映射到或散列到散列表中下标  $i$  的表元素。

 理想散列。考虑一个小城镇的应急系统，其中每个人的电话号码都以 555 开头。令散列函数  $h$  将电话号码转换为它的后 4 位数字。例如，

$$h(555-1264) = 1264$$

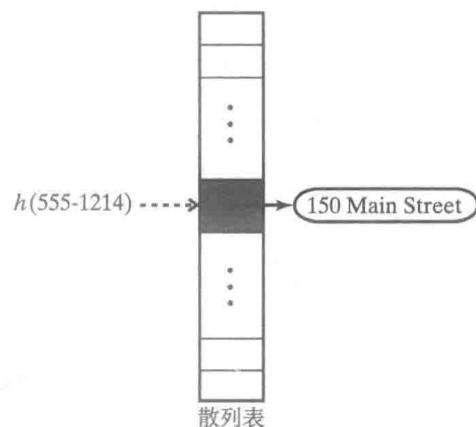
如果 `hashTable` 是散列表，可以将指向与这个电话号码对应的街道地址的引用放在 `hashTable[1264]` 中，如图 22-1 所示。如果计算散列函数的代价很低，则将项添加到数组 `hashTable` 中就是  $O(1)$  操作。

之后要找到对应于号码 555-1264 的街道地址，只需再次计算  $h(555-1264)$ ，使用计算结果去索引 `hashTable`。所以，从 `hashTable[1264]` 中，我们能得到想要找的街道地址。这个操作也是  $O(1)$  的。注意到，我们并没有查找数组 `hashTable`。

将到目前为止我们所掌握的内容，总结为字典添加或获取项操作的简单算法。

```
Algorithm add(key, value)
 index = h(key)
 hashTable[index] = value

Algorithm getValue(key)
 index = h(key)
 return hashTable[index]
```



22.2

22.3

图 22-1 散列函数索引了散列表

这些算法总能奏效吗？如果我们知道所有可能的查找键，它还是起作用的。本例中，查找键范围从 555-0000 到 555-9999，所以散列函数将得到 0 ~ 9999 的索引。如果数组 `hashTable` 有 10 000 个元素，那么每个电话号码将对应 `hashTable` 中的唯一元素。那个元素指向对应的街道地址。这个情景描述了散列的理想情况，而且这里的散列函数称为完美散列函数 (perfect hash function)。

 注：完美散列函数将每个查找键映射为适合作为散列表索引的不同的整数。

 典型散列。在前一个例子中，因为我们需要所有街道地址的数据库，所以每个电话号码都必须对应散列表中的一个项。完美散列函数使得散列表很大，因为它会根据 10 000 个可能的查找键，产生 10 000 个 0 ~ 9999 之间的不同的索引。如果 555 交换机中的每个电话号码都分配了，那么这个散列表总是满的。

虽然对这个应用来说，满散列表十分合理，但大多数散列表都是不满的，甚至可能是稀疏 (sparse) 的，即实际上只使用了很少的元素。例如，如果小镇只需要 700 个电话号码，则 10 000 个位置的散列表中的大部分位置将未使用。我们可能浪费了分配给散列表的大部分空间。如果 700 个号码不是连续的，想使用一个更小散列表，应该需要一个不同的散列函数。

我们可以开发下面这个不同的散列函数。给定非负整数  $i$  和有  $n$  个元素的散列表， $i$  对  $n$  取模的值为从 0 到  $n-1$ 。因为  $i$  是非负的， $i$  模  $n$  是  $i$  除以  $n$  后的整数余数。这个值是散列表

22.4

中的有效地址。所以用于电话号码的散列函数  $h$  可以有下列算法：

```
Algorithm getHashCode(phoneNumber)
// Returns an index to an array of tableSize elements.
i = phoneNumber 的最后四位数字
return i % tableSize
```

这个散列函数——与一般的散列函数一样——执行下列两步：

- 1) 将查找键转换为一个整数，这个整数称为散列码（hash code）。
- 2) 将散列码压缩（compress）到散列表的下标范围内。

查找键常常不是一个整数，通常是一个字符串。所以散列函数首先要将查找键转换为一个整数散列码。下一步，它将这个整数再转换为适合于具体散列表索引的值。

当 `tableSize` 小于 10 000 时，算法 `getHashCode` 描述的散列函数不是完美散列函数。因为 10 000 个电话号码映射到 `tableSize` 个地址，有些电话号码将映射到同一个地址。我们将这种现象称为冲突（collision）。例如，如果

`tableSize` 是 101，那么，`getHashCode("555-1264")` 和 `getHashCode("555-8132")` 都映射到 52。如果已经将 555-1264 的街道地址保存在 `hashTable[52]` 中，如图 22-2 所示，那对 555-8132 的地址将如何处理呢？处理这样的冲突称为冲突解决方案（collision resolution）。在讨论冲突解决方案之前，我们先进一步研究散列函数。

 **注：**一般的散列函数不是完美的，因为它们可能允许多个查找键映射到同一个索引中，这导致散列表的冲突。

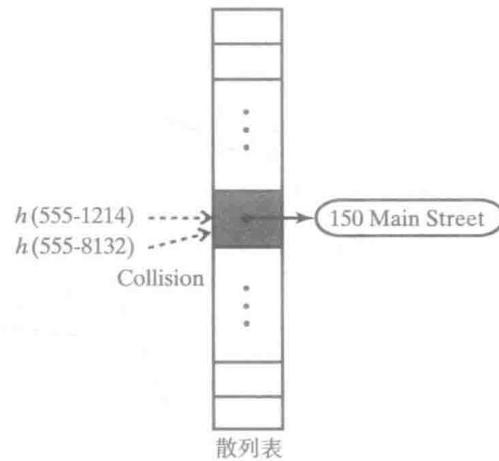


图 22-2 散列函数  $h$  引起的冲突

## 散列函数

**22.5 一般特性。**任何能得到适合作数组下标的整数的函数，都能作为散列函数。但不是每个这样的函数都是好的散列函数。前面的讨论表明好的散列函数应该是

- 具有最少的冲突
- 计算要快

回忆一下，典型散列函数首先将查找键转换为一个整数散列码。然后散列函数再将散列码压缩为一个适合作具体散列表索引的整数。

 **注：**为减少冲突的可能性，选择将项均匀分布到整个散列表的散列函数。

首先，考虑如何将查找键转换为一个整数。要知道查找键可能是基本类型或是类的实例。

## 计算散列码

**22.6 用于类类型的散列码。**Java 的基类 `Object` 有一个方法 `hashCode`，它返回一个整数散列码。因为每个类都是 `Object` 的子类，所以所有的类都继承了这个方法。但除非类重写了 `hashCode`，否则，方法将根据用来调用方法的对象的内存地址返回一个 `int` 值。这个默认

的散列码通常不适合于散列，因为相等但不同的对象会有不同的散列码。为了能将散列码用于字典实现中，散列必须将相等的对象映射为散列表中的同一个位置。所以一个类应该定义自己的 `hashCode` 方法，它遵从下列原则。

### 注：用于方法 `hashCode` 的原则

- 如果类重写了方法 `equals`，它也应该重写 `hashCode`。
- 如果方法 `equals` 认为两个对象相等，则 `hashCode` 必须对两个对象返回相同的值。
- 如果在程序运行期间，多次调用对象的 `hashCode`，且如果此时对象的数据保持不变，则 `hashCode` 必须返回相同的散列码。
- 程序一次执行期间某个对象的散列码，可以不同于同一程序另一次执行期间该对象的散列码。

完美散列函数应该要求，不相等的对象有不同的散列码。但一般地，不相等的对象可能有相同的散列码。因为重复的散列码会导致冲突，你应该想尽办法地避免这种情形。

**用于字符串的散列码。**查找键常常是字符串，所以由一个字符串生成一个好的散列码是很重要的。一般地，先对字符串中的每个字符赋一个整数，然后将这些整数合计形成散列码。例如，可以将整数 1 ~ 26 赋给字母 “A”~“Z”，将整数 27 ~ 52 赋给字母 “a”~“z”。不过，使用字符的 Unicode 值更常见，而且实际上也更容易实现。

假定用于电话号码簿的查找键是名字，如 Brett、Carol、Gail 和 Josh 等。可用几种方法计算这些名字的散列码。例如，可以取每个名字首字母的 Unicode 值来得到不同的散列码。但当几个名字的首字母相同时，如果还使用这个机制，它们的散列码将是一样的。因为出现在名字任何位置的字母都不会有相等的概率，所以使用任何特定字母的散列函数都不会将名字均匀分布到整个散列表中。

假定你将查找键中每个字母的 Unicode 值相加。如果应用中，两个不同查找键永远不会含有相同的字母，则这个方法有效。但含有相同字母的不同排列的查找键——例如，机场代码 DUB 和 BUD 中——将有相同的散列码。这个方法也可能限制了散列码的范围，因为字母的 Unicode 值介于 65 ~ 122 之间。所以这种机制下三字母的词将映射为 195 ~ 366 之间的值。

### 注：现实世界中数据不是均匀分布的。

**用于字符串的更好的散列码。**为字符串生成散列码的更好的方法是，根据每个字符在字符串中的位置，让其 Unicode 值乘上一个因子。然后将这些乘积相加得到散列码。特别地，如果字符串  $s$  有  $n$  个字符， $u_i$  是  $s$  中第  $i$  个字符的 Unicode 值（首字符的  $i$  是 0）。则对某个正常数  $g$ ，生成散列码的公式为

$$u_0g^{n-1} + u_1g^{n-2} + \cdots + u_{n-2}g + u_{n-1}$$

这个表达式是  $g$  的多项式。为了减少算术运算的步数，将多项式写为下列代数等价式：

$$((\cdots((u_0g + u_1)g + u_2)g + \cdots + u_{n-2})g + u_{n-1})$$

计算多项式的这个方法称为 Horner 方法。

下列 Java 语句对字符串  $s$  和 int 常数  $g$  执行这个计算：

```
int hash = 0;
int n = s.length();
```

22.7

22.8

```
for (int i = 0; i < n; i++)
 hash = g * hash + s.charAt(i);
```

字符串的第  $i$  个字符是 `s.charAt(i)`。将这个字符加到乘积 `g * hash` 上，实际上是加上了字符的 Unicode 值。不需要将 `s.charAt(i)` 显式转型为 `int`，而且对结果没有影响。

这个计算可能导致溢出，特别是对于长字符串。Java 忽略这些溢出，而且，如果 `g` 的选择合适，则结果将是一个合理的散列码。目前 Java 语言实现类 `String` 的方法 `hashCode` 时，计算中使用的 `g` 值是 31。但要知道，溢出可能得到一个负数结果。要将散列码压缩为一个合适的散列表索引的工作，由你来完成。



**学习问题 1** `g` 为 31，计算字符串“Java”的散列码。将结果与表达式 `"Java".hashCode()` 的值相比较。

22.9

**用于基本类型的散列码。**本段包含了你可能不熟悉的 Java 操作。不过，对于本章其他内容来讲，这些内容并不是必要的。

如果查找键的数据类型是 `int`，则可以使用键本身作为散列码。如果查找键是 `byte`、`short` 或 `char` 的实例，则可以将它转型为 `int`，从而得到一个散列码。所以，转型为 `int` 是生成散列码的一种方法。

对于其他的基本类型，可以处理它们内部的二进制表示。如果查找键是 `long` 类型的，则它含有 64 位。而 `int` 有 32 位。简单地将 64 位查找键转型为 `int`——或对  $2^{32}$  取模——将会丢失前面的 32 位。结果，只是前 32 位不同的所有键都会有相同的散列码，且发生冲突。由于这个原因，忽略查找键的部分位会出现问题。



**注：**从整个查找键导出散列码。不要忽略任何部分。

不是忽略 `long` 类型查找键中的部分内容，而是将其划分为几段。然后用加法或是像异或（exclusive or）这样的按位布尔操作将它们合在一起。这个过程称为折叠（folding）。

例如，将 `long` 查找键分为两个 32 位的段。为得到左半段，可以将查找键右移若干位。例如，如果我们将 8 位的二进制数 10101100 右移 4 位，则得到 00001010。这样就隔离出数的左半段而丢掉了右半段。现在如果将 00001010 与原来的值合起来并忽略结果的左半部分，事实上就是将原来键的左半部分与右半部分合在了一起。

现在，我们来看看在 Java 中是如何实现这些步骤的。表达式 `key >> 32` 将 64 位的 `key` 右移 32 位，其结果就是去掉了其右半部分。Java 中异或操作是 `^`，它作用于 1 位的结果如下：

`0 ^ 0` 是 0

`1 ^ 1` 是 0

`0 ^ 1` 是 1

`1 ^ 0` 是 1

对于两个多位的值，运算符将每对对应的位拼起来。所以

`1100 ^ 1010` 是 0110

所以，表达式 `key ^ (key >> 32)` 使用异或运算将 64 位 `key` 的两半拼起来。虽然结果是 64 位的，但最右面的 32 位含有 `key` 中两半拼得的结果。将结果转型为 `int`，则会忽略掉最左面的 32 位。所以所需的运算是

```
(int)(key ^ (key >> 32))
```

可以对 `double` 类型的查找键执行类似的运算。因为 `key` 是实数，所以我们不能将它用在前面的表达式中。而是必须调用 `Double.doubleToLongBits(key)` 来得到 `key` 的位串。下列语句得到所需的散列码：

```
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
```

为什么不能简单地将查找键从 `double` 类型转型为 `int` 类型呢？因为查找键是实数，所以将它转型为 `int` 时将只能得到值的整数部分。例如，如果键的值是 32.98，则将它转型为 `int` 得到整数 32。虽然我们可以使用 32 当作散列码，但整数部分是 32 的所有查找键都会将 32 作为散列码。除非你知道各个实数有不同的整数部分，否则将它们转型为 `int` 值会导致很多冲突。

`float` 类型的查找键可以简单地用其 32 位作为散列码。调用 `Float.floatToIntBits(key)` 就可以实现。

基本类型散列码的这些计算，实际上都用在对应的包装类内 `hashCode` 方法的实现中。

## 将散列码压缩为散列表的下标

将一个整数缩放到某个给定范围内的最常见的方法，是使用 Java 中的 `%` 运算符。22.10 对于一个正的散列码  $c$  和一个正整数  $n$ ， $c \% n$  用  $n$  除  $c$ ，余数作为结果。这个余数介于 0 到  $n-1$  之间。所以  $c \% n$  是有  $n$  个位置的散列表的理想下标。

所以  $n$  应该等于散列表的长度，但不是任何的  $n$  都行得通。例如，如果  $n$  是偶数，则  $c \% n$  与  $c$  有相同的奇偶性（parity），即如果  $c$  是偶数，则  $c \% n$  也是偶数；如果  $c$  是奇数，则  $c \% n$  也是奇数。如果散列码偏向于偶数或奇数（注意，基于内存地址的散列码一般都是偶数），则散列表的下标将有相同的偏向性。如果  $n$  是偶数，则下标不是均匀分布的，你会遗漏表中的很多位置。所以  $n$ ——散列表的长度——总应该是奇数。

当  $n$  是素数（prime number）时——只能被 1 和自身整除的数—— $c \% n$  得到 0 到  $n-1$  之间均匀分布的值。素数——除了 2 以外——是奇数。

 **注：**散列表的长度应该是大于 2 的素数。这样，如果你使用  $c \% n$  将正散列码  $c$  压缩到散列表的地址，则地址将在 0 到  $n-1$  之间均匀分布。

还有最后一个细节问题。你之前看到，方法 `hashCode` 可能返回一个负整数，所以需要小心谨慎。如果  $c$  是负数，则  $c \% n$  在  $1-n$  到 0 之间。0 是没问题的，但如果  $c \% n$  是负数，加  $n$  后使得它介于 1 到  $n-1$  之间。

现在可以实现散列函数了。下面的方法计算给定查找键的散列地址，查找键的数据类型是泛型对象类型 `K`。数据域 `hashTable` 是用作散列表的数组。要记得，`hashTable.length` 是数组长度，而不是散列表中当前项的个数。我们假定，这个长度是素数，且方法 `hashCode` 返回与前面的讨论相一致的散列码。22.11

```
private int getHashCode(K key)
{
 int hashIndex = key.hashCode() % hashTable.length;
 if (hashIndex < 0)
 hashIndex = hashIndex + hashTable.length;

 return hashIndex;
} // end getHashCode
```



**学习问题 2** 段 22.8 中的学习问题 1，要求你计算字符串“Java”的散列码。使用那个值来计算当散列表长为 101 时，`getHashCode("Java")` 的返回值。

**学习问题 3** 哪个单字符的串，传给 `getHashCode` 时，能使方法返回的值与前一个学习问题中的值一样？

## 解决冲突

**22.12** 当向字典中添加时，如果散列函数将查找键映射到散列表中一个已经被占用的元素，则需要为查找键的值找到另一个地方。有两种主要选择：

- 选择散列表中的一个未用元素。
- 改变散列表的结构，让每个数组元素可以表示多个值。

找到散列表中一个未用的或开放的元素称为**开放地址法**（open addressing）。这种选择听上去简单，但它会导致几种复杂情形。不过，改变散列表的结构并不像听上去这么困难，可能是比使用开放地址策略更好的解决冲突的方法。我们将考查两种方法，先来看看开放地址法的几种变化形式。



**注：**初始时，散列表中的所有元素都含有 `null`。我们将这些元素看作空的。它们也是开放的，因为可以将字典项放至其中。

### 开放地址的线性探查

**22.13** 当将项添加到散列表而发生冲突时，开放地址策略在散列表中找到一个开放位置作为替代元素。然后使用这个元素指向新的项。

找到散列表中的开放元素称为**探查**（probing），可能有不同的探查技术。对于**线性探查**（linear probing），如果冲突发生在 `hashTable[k]` 处，那么看看 `hashTable[k+1]` 是否可用。如果还不能用，再看看 `hashTable[k+2]`，以此类推。查找过程中考虑的这些散列表位置组成**探查序列**（probe sequence）。如果探查序列到达了散列表尾，则再从散列表的开头继续。所以我们将散列表看作循环的：散列表中的第一个元素紧接在最后一个元素之后。



**注：**线性探查解决散列过程中的冲突的办法是，检查散列表中的连续位置——从原散列地址开始——去查找下一个开放的位置。

**22.14** 相冲突的添加。回忆图 22-2 所示的示例。查找键 555-1264 和 555-8132 都映射到地址 52。假定 555-4294 和 555-2072 也映射到相同的地址，且有下列语句，将项添加到空字典 `addressBook` 中：

```
addressBook.add("555-1264", "150 Main Street");
addressBook.add("555-8132", "75 Center Court");
addressBook.add("555-4294", "205 Ocean Road");
addressBook.add("555-2072", "82 Campus Way");
```

第 1 步的添加将使用 `hashTable[52]` 的位置。第 2 步的添加会发现 `hashTable[52]` 已经被占用，所以它应该向前探查，并使用 `hashTable[53]` 的位置。第 3 步的添加应该发现 `hashTable[52]` 和 `hashTable[53]` 都已经被占用，所以它应该向前探查，并使用 `hashTable[54]` 的位置。最后，第 4 个语句在添加到 `hashTable[55]` 之前，会探查地址为

52、53 和 54 的位置。图 22-3 显示对散列表进行上述添加的结果。

**注：**线性探查可以检查散列表中的每个位置。结果，只要散列表不满，这类探查就可以保证 add 操作的成功。

获取。在添加 4 个项时，已用线性探查解决了冲突，如何获取对应于最后添加的查找键 555-2072 的街道地址呢？即如果执行语句 22.15

```
String streetAddress = addressBook.getValue("555-2072");
```

则 getValue 会如何动作？因为 getHashCode("555-2072") 是 52，则 getValue 将查找数组中从 hashTable[52] 开始的连续元素，直到找到对应于查找键 555-2072 的街道地址为止。且慢！我们如何告之哪个街道地址是正确的那个？我们不能，除非将查找键与其值打包在一起。第 21 章 21.2 段提供了类 Entry，我们可用在此处。图 22-4 显示对图 22-3 所给的散列表做了这个修改后的结果。

现在，查找 555-2072 时，可以沿着将这个查找键和值添加到散列表中时使用的相同的探查序列进行。当稍后我们评估散列方法的效率时，会用到这个事实。

**注：**成功查找对应于给定查找键的项，沿着将项添加到散列表中的相同的探查序列进行。

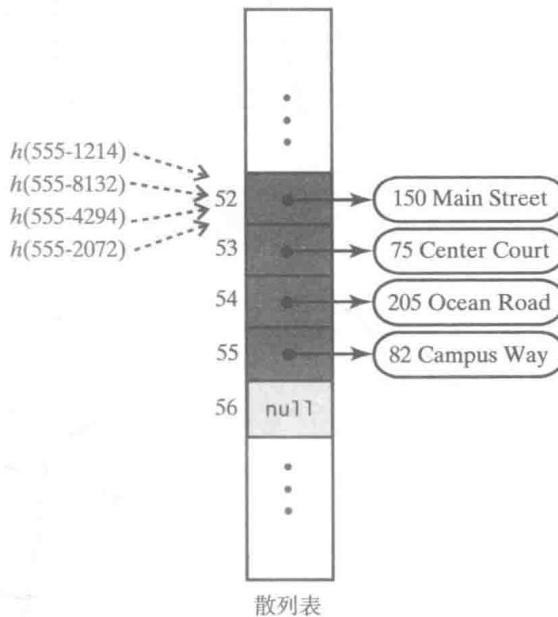


图 22-3 添加其查找键散列到同一地址的 4 项后，线性探查的效果

如果查找键不在散列表中将会怎样？探查序列的查找总会遇到 null 位置，表示查找不到。但在得出这个结论之前，必须知道 remove 方法是如何做的，因为它可能对后续的获取有不利的影响。

删除。假定如图 22-4 所示添加了 4 次后，我们执行下列代码删除两项： 22.16

```
addressBook.remove("555-8132");
addressBook.remove("555-4294");
```

从散列表中删除项的最简单方法是用 null 替换这个项。图 22-5 显示的散列表，是 remove 方法将 null 放到数组元素 hashTable[53] 和 hashTable[54] 后的情况。但现在尝试查找查找键 555-2072，会中止于 hashTable[53]，查找不到。虽然散列表中从未用过的一个元素应该令查找中止，但曾经用过但现在又可用的元素不应该有这样的作用。

所以，我们需要区分散列表中的 3 类元素：

- 占用的——指向字典中一个项的元素
- 空的——含有 null 且永远含有这个值的元素
- 可用的——该元素的项已经从字典中删除

空的或可用的元素都是开放的。相应地，方法 remove 不应该用 null 来替换项，而是应该将项的这个位置标记为可用的。获取时，如果查找遇到一个可用元素，则应该继续查

找，并且仅当查找成功或者到达 `null` 时查找才应该停止。删除时的查找也是如此。

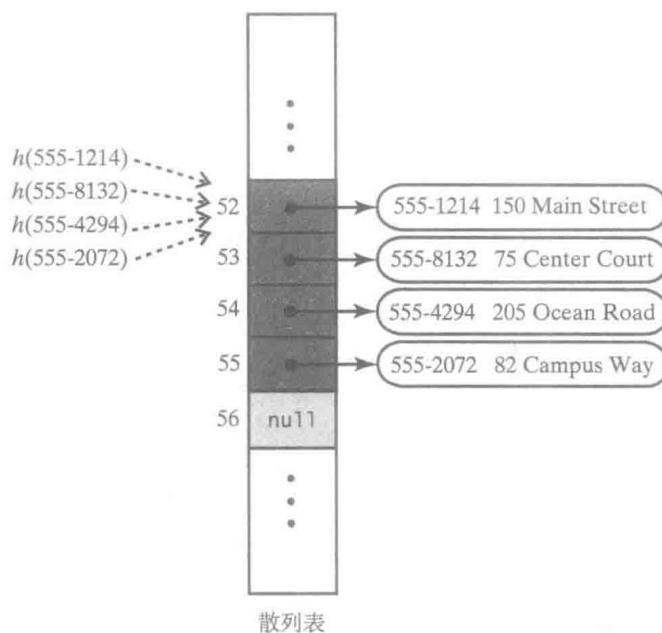


图 22-4 修改图 22-3 所示的使用线性探查解决冲突的散列表；每个项包含一个查找键和其对应的值

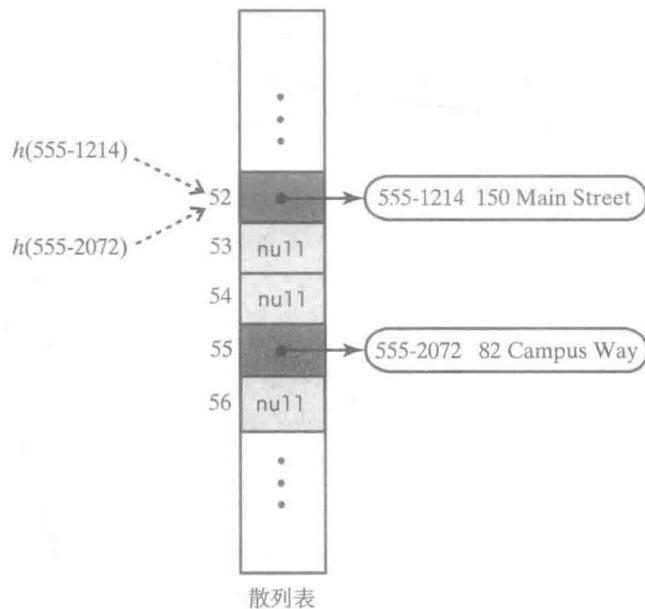


图 22-5 如果 `remove` 使用 `null` 来删除项的散列表



**学习问题 4** 给出能表示散列表中元素的 3 种状态的实现方法。这个状态应该属于元素还是应该属于它指向的字典项？

22.17

添加过程中散列表中元素的重用。回忆图 22-4 所示的散列表。查找键是 555-2072 的项映射到 `hashTable[52]` 中，但因为冲突，最终添加到散列表的 `hashTable[55]` 中。图 22-6a 以更简单的方式再次显示了这个散列表。4 个已占据的元素构成探查序列；其他的

位置含有 null。因为查找键 555-2072 映射到探查序列的第一个位置，但实际占据第 4 个位置，使用简单的顺序查找就能找到它。

现在，我们试着删除探查序列中间的两项，如图 22-6b 所示。对 555-2072 的查找从探查序列的头开始，必须继续探查到被删项的后面，在探查序列的最后一个元素成功找到，然后停止。如果 555-2072 没有出现在这个最后元素中，则查找失败于下一个元素，因为它含有 null。图 22-6c 说明了这个查找过程。

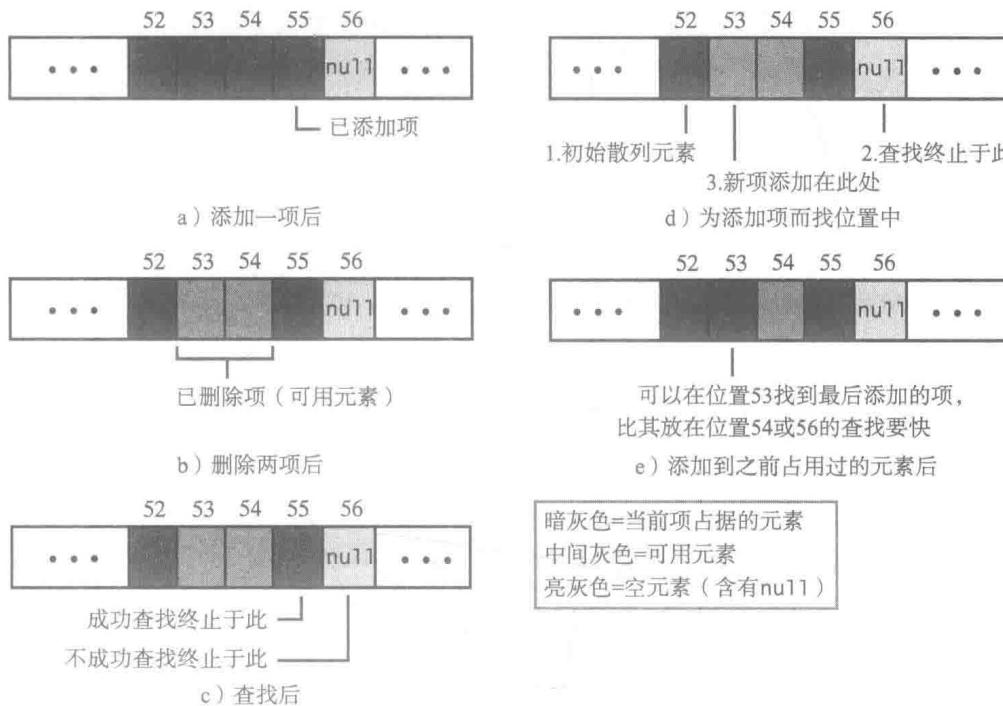


图 22-6 不同情况下的线性探查序列

最后，考虑添加映射到这个探查序列的项时会发生什么事情。例如，查找键 555-1062 映射到 `hashTable[52]` 中。`add` 操作必须先查看这个查找键是否已在散列表中。为此，它查找探查序列。必须查找全部探查序列，并到达含 `null` 的元素时，发现 555-1062 没有在散列表中。图 22-6d 显示这次查找结束于 `hashTable[56]`。`add` 会将新项放在这个位置吗？它能放在那儿，但如果 `add` 重用表中目前呈可用状态的元素，则这样的添加过程会更快些。这样的两个元素在下标 53 和 54 处。我们应该将新项放在 `hashTable[53]` 中——即最接近探查序列开头的位置——这样以后的查找会更快。这次添加后的散列表如图 22-6e 所示。



**注：**当使用开放地址法解决冲突时散列表所需的查找过程

- 获得操作在探查序列中查找 `key`。它检查当前项，忽略呈可用状态的元素。当找到 `key` 或是到达 `null` 时，查找停止。
- 删除操作使用与获取同样的逻辑查找探查序列。如果它找到 `key`，则将这个元素标记为可用的。
- 插入操作使用与获取同样的逻辑查找探查序列，但它还要记下遇到的第一个为可用状态或是含有 `null` 的元素下标。如果没有找到 `key`，则操作将记下的这个元素用于新项。在探查过程中我们记下这个下标。

**22.18 私有方法 probe。**方法 `probe(key, index)` 从 `hashTable[index]` 开始，沿着探查序列查找 `key`。查找过程忽略呈可用状态的元素，并继续查找，直到到达含有 `key` 或是 `null` 的位置。在查找过程中，如果散列表中有被删除的项，则方法记录下指向第一个项的元素下标。故 `probe` 返回一个表元素下标，这个元素或者是指向含有 `key` 项的引用，或者是能进行添加的位置。

注意到，`probe` 返回的是沿探查序列查找时遇到的第一个可用元素的下标。因为 `add` 将新项放到这个元素中，所以后序查找这个项时，比起 `add` 将它放到沿探查序列更远的元素中，来得更快些。

下列伪代码描述了 `probe` 的逻辑。

```
Algorithm probe(index, key)
// Searches the probe sequence that begins at index. Returns the index of either the element
// containing key or an available element in the hash table.
```

```
while (没找到key且hashTable[index]不是null)
{
 if (hashTable[index]指向字典中的一个项)
 {
 if (hashTable[index]中的项含有key)
 退出循环
 else
 index = 下一个探查下标
 }
 else // hashTable[index] is available
 {
 if (这是遇到的第一个可用元素)
 availableStateIndex = index
 index = 下一个探查下标
 }
}
if (找到key或者没有遇到一个可用元素)
 return index
else
 return availableStateIndex // Index of first entry removed
```

下列方法实现了这个线性探查算法。

```
// Precondition: checkIntegrity has been called.
private int linearProbe(int index, K key)
{
 boolean found = false;
 int availableStateIndex = -1; // Index of first element in available state
 while (!found && (hashTable[index] != null))
 {
 if (hashTable[index] != AVAILABLE)
 {
 if (key.equals(hashTable[index].getKey()))
 found = true; // Key found
 else
 // Follow probe sequence
 index = (index + 1) % hashTable.length; // Linear probing
 }
 else // Element in available state; skip it, but mark the first one encountered
 {
 // Save index of first element in available state
 if (availableStateIndex == -1)
 availableStateIndex = index;
 index = (index + 1) % hashTable.length; // Linear probing
 } // end if
 } // end while
```

```
// Assertion: Either key or null is found at hashTable[index]
if (found || (availableStateIndex == -1))
 return index; // Index of either key or null
else
 return availableStateIndex; // Index of an available element
} // end linearProbe
```

**聚集。**使用线性探查解决冲突，导致散列表中一组组连续的元素被占用。每一组称为一个簇（cluster），这种现象称为基本聚集（primary clustering）。实际上，每个簇是添加、删除或是获取一个表项时必须查找的一个探查序列。当没有太多的冲突发生时，探查序列很短，且能快速查找。但当添加时，簇内发生的冲突又会增加簇的长度。簇越大意味着冲突下查找时间越长。当簇变长时，它们能合并为更大的簇，使问题更严重。这种现象可能使散列表的一部分放置很多的项，而另一部分相对较空。

22.19

 **注：**线性探查易于导致基本聚集。每个簇是散列表中一组连续占用的位置。添加时，簇中任何位置上的任何冲突又导致簇更长。

## 开放地址的二次探查

通过改变用来解决冲突时的探查序列，可以避免基本聚集。正如前一节所讨论的，如果给定的查找键散列到地址  $k$ ，则线性探查将查找从地址  $k$  开始的连续表元素。另一方面，二次探查（quadratic probing），考虑下标为  $k+j^2$  ( $j \geq 0$ ) 的元素，即它使用地址  $k, k+1, k+4, k+9, \dots$ ，依此类推。与之前一样，如果探查序列到达散列表尾，则它会绕回到表的开头。这个开放地址策略下，探查序列中前两项之后的各项不是连续的。事实上，这个距离增量越到序列的后面越大。图 22-7 中，标记出了组成散列表中这样一个探查序列的 5 个位置。

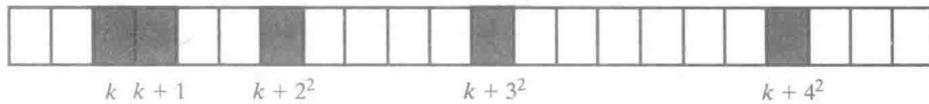


图 22-7 使用二次探查的长度为 5 的探查序列

除了改变探查序列这一点，二次探查与线性探查是一样的。它使用段 22.16 中描述的三个状态：占用的、空的和可用的。另外，它重用表中为可用状态的表元素，如段 22.17 所描述的。

虽然二次探查避免了基本聚集，但与表中已有的项发生冲突的项，会使用相同的探查序列，所以增加了探查序列的长度。这种现象——称为二级聚集（secondary clustering）——通常不是严重的问题，但它增加了查找时间。

线性探查的优点是，它能到达散列表中的每个元素。正如我们之前所说的，这个特点很重要，因为它能保证当散列表不满时 add 操作能够成功。二次探查也能保证 add 操作的成功，只要散列表至多是半满的，且其长度是素数。（见本章末的练习 8。）

计算探查序列的下标时，二次探查比线性探查要花更多的时间。本章末的练习 2 显示如何高效地计算这些下标。

22.20

 学习问题 5 执行二次探查而不是线性探查时，方法 linearProbe 必须做哪些修改？



注：二次探查

- 通过检查散列表中最初的散列索引加上 $j^2$  ( $j \geq 0$ ) 处的元素，解决散列过程中的冲突。
  - 如果表长是素数，则能到达散列表中一半的元素。
  - 避免基本聚集但可能导致二级聚集。

## 开放地址的双散列

从最初的散列索引  $k$  开始，线性探查和二次探查都在  $k$  上加上一个增量来定义探查序列。这些增量——对于线性探查是 1，对于二次探查是  $j^2$ ——不依赖于查找键。双散列 (double hashing) 使用第二个散列函数以依赖于查找键的方式来计算这些增量。通过这种方式，双散列避免了基本聚集和二级聚集。

与其他开放地址策略一样，双散列应该得到一个能到达整个表的探查序列。如果散列表长是素数，情况就是这样的。（见本章末的练习 9。）第二个散列函数必须不同于最初的散列函数，且永远不能有 0 值，因为 0 不是合适的增量。

示例。例如，对于长度为 7 的散列表，考虑下列散列函数对：



$$h_1(\text{key}) = \text{key \% } 7$$

$$h_2(\text{key}) = 5 - \text{key} \% 5$$

这个散列表非常小，但它能让我们研究探查序列的行为。查找 16 时，我们有：

$$h_1(16) = 2$$

$$h_2(16) = 4$$

如果我们将表长变为 6，且使用下面的两个散列函数，会发生什么情况？

$$h_1(\text{key}) = \text{key \% } 6$$

$$h_2(\text{key}) = 5 - \text{key \% } 5$$

查找键为 16 时，有

$$h_1(16) = 4$$

$$h_2(16) = 4$$

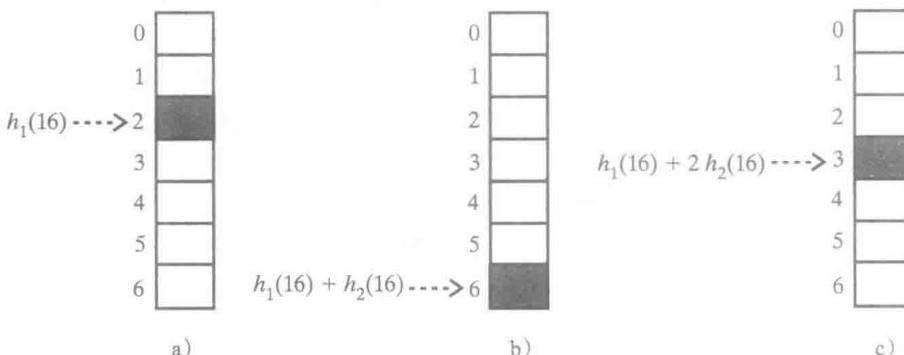


图 22-8 查找键为 16 时由双散列产生的探查序列中的前三个位置

注：双散列

- 通过检查散列表中最初的散列地址加上由第二个散列函数定义的增量的元素，解决散列过程中的冲突。第二个散列函数应该
    - ◆ 不同于第一个散列函数
    - ◆ 依赖于查找键
    - ◆ 有非零值
  - 如果表长是素数，则能到达散列表中的每个元素
  - 避免基本聚集和二级聚集

注：修改散列函数

段 22.11 中定义的散列函数 `getHashIndex` 提供了我们开始探查散列表的下标。回忆一下，探查查找给定的查找键，忽略从中删除了项的表元素。查找继续，直到到达查找键或是 `null` 的位置。在需要探查时会用到我们非正式提出的下列私有方法。

```

Algorithm probe(index, key): integer
// Detects whether key collides with hashTable[index] and resolves it by following a probe
// sequence.
// Returns the index of an element along the probe sequence that is either available or
// contains the entry whose search key is key. This index is always legal, since the probe
// sequence stays within the hash table.
// Precondition: The dictionary is initialized.
 availableIndex = -1 // Index of first element from which an entry had been removed
 while (没找到key且hashTable[index]不是null)
 {
 if (hashTable[index]不是可用的)
 {
 if (key.equals(hashTable[index].getKey()))
 退出循环 // Key found
 else
 // Follow probe sequence
 index = (index + 1) % hashTable.length; // Linear probing
 }
 else
 // Skip entries that were removed
 {
 // Save index of first element from which an entry had been removed
 if (availableIndex == -1)
 availableIndex = index;
 index = (index + 1) % hashTable.length; // Linear probing
 }
 }
 // Assertion: Either key or null is found at hashTable[index]
 if (找到key或是availableIndex == -1)
 return index // Index of either key or null
 else
 return availableIndex // Index of an available element

```

为了找到正确的散列索引，探查是必需的步骤，所以将它添加到 `getHashCode` 中，如下所示。

```
private int getHashCode(K key)
{
 int hashIndex = key.hashCode() % hashTable.length;
 if (hashIndex < 0)
```

```

hashIndex = hashIndex + hashTable.length;
hashIndex = probe(hashIndex, key);
return hashIndex;
} // end getHashIndex

```

我们将方法 `probe` 的定义留作练习。



**学习问题 6** 采用双散列法，当散列函数如下时，散列表的长度应该是多少？为什么？

$$h_1(\text{key}) = \text{key \% } 13$$

$$h_2(\text{key}) = 7 - \text{key \% } 7$$

**学习问题 7** 当查找键是 16 时，前一问给出的散列函数定义的探查序列是什么？

## 开放地址的潜在问题

22.23

前三种用于解决冲突的开放地址策略，均假定每个表元素处于三种状态之一：占用的、空的或可用的。回忆一下，仅有空元素含有 `null`。频繁的添加和删除可能导致散列表中的每个元素都指向当前项或是以前的项。即散列表可能没有含 `null` 的元素，不管目前位于字典中的项是多还是少。如果发生这种情况，那么查找探查序列的方法将没法工作。而是，每次查找都要检查散列表的每个元素之后才失败。另外，结束查找的检查有些复杂，且比查找 `null` 更费时。

你应该保证你的实现不要出现这种失败。增大散列表的长度（见第 23 章的段 23.7）可以修正这个问题，因为新表的元素含有 `null`。拉链法——下面要讨论的——不会出现这个问题。



**注：**当应用不需要删除时，开放地址法可以进行简化。例如，编译器建立符号表时就是这样的情况。它可以使用不允许删除的字典。散列表中的元素将指向一个字典项，或是含有 `null` 值。段 22.16 中给出的三种状态的定义不是必需的。本章结尾的项目 1 要求你来实现这个字典。

## 拉链法

22.24

解决冲突的第二种常规方法是改变散列表的结构，以便每个元素可以表示多个值。这样的一个元素称为一个桶（bucket）。当新的查找键映射到某个元素时，只需将键和其对应的值放到桶中，很像是在开放地址中的做法。为了找到一个值，先散列查找键，然后找到桶，在桶中查找键-值对。多半情况下，桶中含有很少的几个值，所以这种微查找会很快。当你删除一项时，在其桶中找到并删除它。所以项不再属于散列表。

什么结构能用来表示一个桶？你熟悉的线性表、有序表、结点链表、数组或是向量都有可能。任何使用数组或是向量的结构都会导致大量的内存开销，因为散列表中的每个桶都要分配固定大小的内存。这些内存中的大部分都没有使用。线性表的链式实现，或是结点链表是桶的合理选择，因为可按需给桶分配内存。图 22-9 说明的是使用链表作为

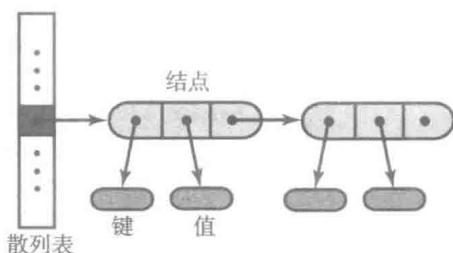


图 22-9 使用拉链法的散列表；每个桶都是一个结点链表

桶的散列表。这种结构中，散列表中的每个元素是指向组成桶的结点链表的头引用。每个结点含有指向查找键的引用、指向键对应值的引用和指向链表中下一个结点的引用。注意到，一个结点必须指向查找键，这样以后查找链表时才能找到它。使用链式桶的冲突解决方法称为拉链法 (separate chaining)。

如果你的字典允许查找键重复，则将新项添加到对应链的头是最快的，如图 22-10a 所示。但是，如果想让查找键唯一，则添加新项时要求你在链表中查找查找键。如果没有找到，会到达链表表尾，这是添加新项的地方。图 22-10b 说明了这种情形。但因为你必须查找链表，所以链表要按查找键有序，如图 22-10c 所示。这样后续的查找才会更快点。不过如你所见，一般的链表都很短，所以这种改进不会浪费太多时间。

22.25

### 学习问题 8 考虑查找键为不同的整数。如果散列函数是

$$h(\text{key}) = \text{key \% } 5$$

用拉链法来解决冲突，则添加下列查找键后，它们应该在散列表的什么位置？

4, 6, 20, 14, 31, 29

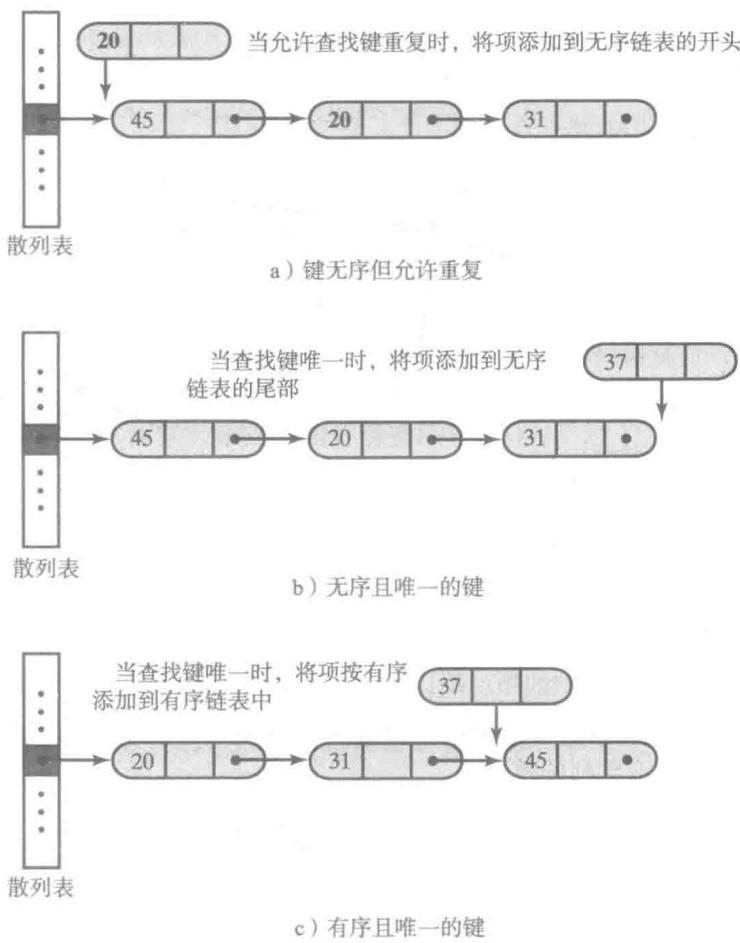


图 22-10 根据整数查找键的特性，将新项插入链式桶中

对于不重复的查找键和无序链表，字典中 `add`、`remove` 和 `getValue` 方法的算法如下所示。

22.26

```

Algorithm add(key, value)
index = getHashCode(key)
if (hashTable[index] == null)
{
 hashTable[index] = new Node(key, value)
 numberofEntries++
 return null
}
else
{
 查找链，从hashTable[index]开始，查找含有key的结点
 if (找到key)
 { // Assume currentNode references the node that contains key
 oldValue = currentNode.getValue()
 currentNode.setValue(value)
 return oldValue
 }
 else // Add new node to end of chain
 { // Assume nodeBefore references the last node
 newNode = new Node(key, value)
 nodeBefore.setNextNode(newNode)
 numberofEntries++
 return null
 }
}

Algorithm remove(key)
index = getHashCode(key)
查找链表，从hashTable[index]开始，查找含有key的结点
if (找到key)
{
 从链表中删除含有key的结点
 numberofEntries--
 return 被删除结点中的值
}
else
 return null

Algorithm getValue(key)
index = getHashCode(key)
查找链表，从hashTable[index]开始，查找含有key的结点
if (找到key)
 return 找到的结点中的值
else
 return null

```

所有这三个操作都要查找结点链表。如果散列表足够大且散列函数将项均匀分布在表中，每个链表都应该仅含有很少的项。所以这些操作应该是时间高效的。对于含  $n$  个项的字典，操作当然要快于  $O(n)$ 。但最差情况下，所有的项都在一个链表中，故效率退化为  $O(n)$ 。我们将在第 23 章详细讨论散列的效率。



**注：**拉链法提供了高效且简单的冲突解决方案。但因为改变了散列表的结构，所以拉链法比开放地址需要更多的内存。



**学习问题 9** 对于不重复的查找键和有序拉链，写出字典 add 方法的算法。

**学习问题 10** 当使用散列实现字典时，能定义查找键的按序迭代吗？请解释。

## 本章小结

- 散列是将键 – 值项保存到称为散列表的数组中的一项技术。散列函数将项的查找键转换为将含有该项的数组元素的下标。
- 所有的类都有方法 `hashCode`，它返回一个整数值散列码。如果类的实例要作为查找键，则你应该重写 `hashCode` 以得到合适的散列码。散列码应该依赖于整个查找键。
- 散列函数使用 `hashCode` 从查找键来计算散列码，然后将散列码压缩为散列表的下标。压缩散列码  $c$  的典型方法是计算  $c$  模  $n$ ，其中  $n$  是素数，且为散列表的表长。这个计算得到其值介于 0 到  $n-1$  之间的下标。
- 完美散列函数将每个查找键映射到散列表中唯一的元素。如果你知道所有可能的查找键，则可以找到这样一个函数。使用完美散列函数能使字典操作达到  $O(1)$  的。
- 对于一般的散列函数，多个查找键可能映射到散列表中的同一位置。这种现象称为冲突。
- 可用不同的方法来处理冲突。其中有开放地址法和拉链法两种。
- 对于开放地址法，映射到同一位置的所有项最终都保存在散列表中。这些项保存在称为探查序列的元素序列中。常见有几种不同的开放地址方法。线性探查使用连续的元素。二次探查中，探查序列中元素的增量越来越大。这些增量是 1, 4, 9, 等等，即整数 1, 2, 3…的平方。双散列使用一个依赖于查找键的固定的增量。第二个散列函数提供这个增量。
- 对于开放地址法，将放置项的散列表元素标记为可用状态就可以删除项。不能将它的元素置为 `null`，因为这会过早地中止后续的查找。通过查找探查序列可以获取项，忽略可用元素，直到找到所需的项或是遇到 `null` 时为止。当添加新项时也执行相同的查找，但查找时，要标记出第一个可用元素——如果有——它指向一个已删除的项。这个元素用来保存新添加的项。如果没有这样的元素，则使用查找整个序列之后遇到的 `null` 元素，添加会加长探查序列。
- 线性探查和二次探查的缺点是聚集。聚集加长了探查序列，所以会增加查找它的时间。双散列避免了这个问题。
- 对于拉链法，散列表是桶的数组。映射到同一表元素的所有项都保存在那个位置指向的桶中。比如，每个桶可以是一个结点链表。即散列表中的每个元素都指向链表表头。
- 你可以将新项按查找键有序添加到桶的链表中。虽然有序链表可以改善查找时间，但它们通常也不是必要的，因为一般的链表都很短。如果允许重复键，则将新项添加到无序链表的表头，如果不允许，则添加到链表表尾。
- 对于拉链法，通过将查找键映射到表元素，然后查找那个位置所指向的桶，可以获取或删除一个项。
- 整个散列表的迭代不是有序的，即使使用有序桶拉链法来解决冲突时。

## 练习

1. 为附录 B 段 B.16 中给出的类 `Name` 定义一个 `hashCode` 方法。
2. 二次探查使用下列下标来定义探查序列：

$$(k+j^2) \% n \quad j \geq 0$$

其中  $k$  是散列下标,  $n$  是散列表长。

- a. 如果散列表含有 17 个位置, 散列下标是 3, 则二次探查定义的探查序列中前 6 个数组元素下标是什么?

- b. 你可以使用递推关系, 来高效地计算探查序列的下标

$$k_{i+1} = (k_i + 2i + 1) \% n \quad i \geq 0 \text{ 且 } k_0 = k$$

试推导这个递推关系。

- c. 说明你可以用比较操作及少量的减法, 替代 b 中的取模运算。

3. 第 21 章中的项目 10 修改了程序清单 21-1 中的 `Entry` 类, 让它变为公有的, 然后用它来实现字典。考虑 `Entry` 对象的散列表。不再修改 `Entry` 的定义, 如何表示被删除的项?

4. 假定散列表长是 31, 用段 22.8 中描述的散列码, 使用拉链法解决冲突。列出散列到表中同一元素的 5 个不同的名字。

5. 假定有练习 4 中描述的散列表和散列函数, 但使用开放地址法的线性探查解决冲突。列出不全部散列到表中同一元素, 但会导致冲突及聚集的 5 个不同的名字。

6. 重做练习 5, 但使用开放地址法的二次探查解决冲突。

7. 给出即使表长是素数也不能到达整个散列表的一个二次探查产生的探查序列示例。

8. 说明, 如果散列表最多有一半是满的, 且表长是素数, 则二次探查将能保证成功添加。

9. 说明, 如果表长是素数, 则双散列能得到到达整个表的探查序列。提示: 说明如果增量和表长是互素, 则这个结论是正确的。然后, 如果表长是素数, 则所有增量都与它互素。

10. 假定你按照下面所说的修改段 22.13 中的线性探查机制。当在 `hashTable[k]` 发生冲突时, 检查 `hashTable[k+c]`、`hashTable[k+2*c]`、`hashTable[k+3*c]`, 以此类推, 这里  $c$  是个常量。这个机制能消除基本聚集吗?

11. 段 22.18 定义了方法 `linearProbe`。定义方法 `doubleProbe`, 使用双散列执行探查。

12. 考虑查找键含有 3 个浮点值(例如经度、纬度和海拔)的数据。为这个数据提出至少两个可能的散列函数。

13. 想将大约 1000 个极小的图像保存在使用散列实现的字典中。每个图像是 20 像素宽乘 20 像素高, 每个像素是 256 种颜色之一。提出可能使用的散列函数。

## 项目

1. 段 22.22 结尾处的注中, 描述了不支持删除操作的字典。使用开放地址法用线性探查解决冲突, 实现这个字典。
2. 考虑医疗机构中的病人记录。每个记录含有一个用于病人的整数标识符, 及日期、就诊原因和治疗处方等字符串。设计并实现类 `PatientRecord`, 让其重写 `hashCode` 方法。写程序来测试这个新的类。
3. 设计类 `PatientDataBase`, 保存前一个项目中描述的 `PatientRecord` 的实例。这个类应该提供最少 3 个查询操作, 如下所示。给定一个病人标识符及日期, 第一个操作应该返回就诊原因, 第二个操作应该返回处方。第三个查询操作对给定的病人标识符, 应该返回日期列表。
4. 下列实验比较线性探查和二次探查的性能。你需要一个含有 500 个名字的列表, 或从你的老师那里或从系统管理员那里获得。实现长度为 1000 的散列表, 使用段 22.8 描述的散列码。统计将 500 个名字添加到散列表中时线性探查和二次探查发生的冲突次数。表长分别为 950, 900, 850, 800, 750, 700, 650 和 600 时重复这个实验。
5. 设计并实现类似于项目 4 中的项目, 但不是比较线性探查和二次探查, 而是比较两个不同的散列函数。

6. 写一个程序，使用散列来推断用户做的两个选择。下列输出示例说明计算机和使用者之间的交互：

```

Choose either A or B, and I will guess your choice.
Press Return when you are ready.
I guess that you chose A; am I right? no
Score: 0 correct, 1 incorrect

Choose either A or B, and I will guess your choice.
Press Return when you are ready.
I guess that you chose A; am I right? yes
Score: 1 correct, 1 incorrect

Choose either A or B, and I will guess your choice.
Press Return when you are ready.
I guess that you chose B; am I right? yes
Score: 2 correct, 1 incorrect

...

```

初始时，程序将随机做个猜测。用户进行 5 次选择后，程序应该开始建立一个散列表，用它来预测未来的选择。用户的最后 4 个选择组成散列表的键。保存在表中散列地址的值表示用户下一次选择是 A 的次数，及是 B 的次数。程序使用这个计数来做猜测。

例如，如果 AAAB 散列到含有计数 5 和 2 的对象，其中 5 是用户在已经选择了 A, A, A 和 B 之后选择 A 的次数，程序应该预测 A 是用户的下一个选择。说明，你的程序如何基于提供给你的这些计数来做这个预测。

7. 散列用于计算机及通信的安全应用中。美国政府和其他机构为这些场景已经开发了一系列安全散列算法（Secure Hash Algorithm, SHA）。它们没有冲突，且不可逆，通常被称为消息摘要。这些散列函数中有一个是 SHA-512。当任意长度的一个字符串作为位序列传给 SHA-512 散列函数时，得到的结果总是 512 位的。所以这个散列函数将任意字符串映射为  $2^{512}$  种不同值之一。

正如在本章中看到的散列函数一样，输入中小小的改变也会大大地改变散列函数的结果。当验证信息的内容或是数据集没有改变时，这个改变就是有用的。可以将信息位（通信安全）或是保存在磁盘中的数据（数据取证）传给散列函数，以得到散列码。在发送消息或再次读取磁盘驱动器之后，将得到的位进行散列。如果产生了相同的散列码，则接收者知道原始信息或是磁盘数据没被改变。

因为这类散列函数不可逆，所以可用来保存密码。开发一个应用程序，用来保存用户密码作为散列码及用户名字。当用户登录时，散列密码并在散列表中进行查找，以验证用户。

可以先创建 `java.security.MessageDigest` 的实例，然后散列这个字符串，如下所示。

```

// Create the MessageDigest object
MessageDigest myDigest = MessageDigest.getInstance("SHA-512");
// Update the object with the hash of 'someStringToHash'
myDigest.update(someStringToHash.getBytes());
// Get the SHA-512 hash from the object
byte hashCode[] = myDigest.digest();

```

`digest` 方法返回一个表示 `someStringToHash` 的散列的字节数组。你可以以这种形式使用这个散列值，也可能想将它转为十六进制显示出来。

```

// Convert byte array to hex for display purposes
StringBuffer hexHash = new StringBuffer();
for (int i = 0; i < hashCode.length; i++)
{
 String hexChar = Integer.toHexString(0xff & hashCode[i]);
 if (hexChar.length() == 1)
 {
 hexHash.append('0');
 } // end if
 hexHash.append(hex);
} // end for
System.out.println("Hex format : " + hexHash.toString());

```

# 使用散列实现字典

**先修章节：**第 4 章、第 11 章、第 12 章、Java 插曲 4、第 20 章、第 21 章、第 22 章  
**目标**

学习完本章后，应该能够

- 描述不同的冲突解决方案的相对效率
- 描述散列表的装填因子
- 描述再散列及为什么它是必要的
- 使用散列来实现 ADT 字典

第 22 章说明了当查找是主要工作时，用来实现字典的散列技术。现在研究散列的性能，并讨论在 Java 中的实现细节。

## 散列的效率

**23.1** 正如在第 22 章中所见，ADT 字典的实现依赖于字典中查找键是否唯一。本节，我们仅讨论查找键唯一的情况。回忆一下，这种字典中的 `add` 方法必须要保证不能出现重复查找键的情况。

每种字典操作 `getValue`、`remove` 和 `add` 都要在散列表中查找给定的查找键。对给定键的查找成功或失败，直接影响获取及删除操作的成功或失败。新项的成功添加仅在对给定键查找失败之后才会发生。不成功的添加使用新项的值替换已有项的值。这个操作发生在对给定键的成功查找之后。所以，对这些操作的时间效率有下列结论：

- 成功获取或删除，与成功查找有相同的效率
- 不成功获取或删除，与不成功查找有相同的效率
- 成功添加与不成功查找有相同的效率
- 不成功添加与成功查找有相同的效率

基于这些结论就可以分析在散列表中查找给定查找键的时间效率了。



**注：**成功查找一个项时要查找的链表或探查序列，与在最初将它添加到散列表时所查找的链表或探查序列相同。所以，对一个项的成功查找代价，与插入这个项的代价是一样的。

## 装填因子

**23.2** 第 22 章讨论散列是从不会导致冲突的完美散列函数开始的。如果你对特定的查找键集能找到一个完美的散列函数，则用它来实现 ADT 字典时，会让每个操作都是  $O(1)$  的。这样的实现是理想的。好消息是，找到一个完美散列函数在特定情形下确实可行。但不幸的是，使用完美散列函数并不总是可能的或实际的。这些情形下，多半会发生冲突。

解决冲突要花时间，所以会使字典操作慢于  $O(1)$  操作。随着散列表渐满，冲突会发生

得更频繁，进一步降低了性能。因为解决冲突比计算散列函数的时间要多很多，所以散列开销中最主要的部分是解决冲突。

为了更好地表示这个开销，我们定义如何度量散列表有多满。这个度量——装填因子 (load factor)  $\lambda$ ——是字典长度与散列表长度之比。即

$$\lambda = \frac{\text{字典项数}}{\text{散列表中的位置数}}$$

注意，当字典——因此也是散列表——为空时  $\lambda$  为 0。 $\lambda$  的最大值依赖于你使用的冲突解决类型。对于开放地址机制，当散列表满时， $\lambda$  的最大值是 1。那种情形下，字典中的每个项使用散列表中的一个元素。注意到，可用状态元素的个数不影响  $\lambda$  值。对于拉链法，字典中项的个数可以超出散列表的大小，所以  $\lambda$  没有最大值。

### 注：装填因子

装填因子  $\lambda$  用来衡量解决冲突的代价。它是字典中项的个数与散列表长度之比。 $\lambda$  永远不会为负数。对于开放地址法， $\lambda$  不会超过 1。对于拉链法， $\lambda$  没有最大值。正如你将看到的，限制  $\lambda$  的大小会改进散列的性能。

 学习问题 1 开放地址法中当  $\lambda$  是 0.5 时，散列表中有多少个元素含有字典项？

学习问题 2 对于拉链法， $\lambda$  能不能表示散列表中有多少个桶不是空的？请解释。

## 开放地址法的代价

回忆一下，所有开放地址机制中，字典中的每个项都使用散列表中的一个元素。字典的每个操作 `getValue`、`remove` 和 `add` 都需要查找探查序列，而该序列实际上由查找键和实际使用的冲突解决机制确定。分析这些查找的效率就足够了。

对于我们之前讨论过的每个开放地址机制，我们将说明在散列表中查找查找键时所需要的比较次数。我们用装填因子  $\lambda$  来表示这些数。这些数的推导有些麻烦，有时还是困难的，所以我们忽略这些。但会明确地解释结果。回忆一下，对于开放地址法， $\lambda$  的范围为 0 ~ 1，前者对应散列表是空的，后者对应散列表是满的。

**线性探查。** 使用线性探查情况下，当散列表渐满时可能会发生更多的冲突。冲突后，查找组成簇的探查序列。如果添加了新项，则簇的长度增大。所以可以预料到探查序列也增大，故需要更长的查找时间。事实上，对于给定的查找键，查找探查序列时比较的平均数约为

$$\frac{1}{2} \left\{ 1 + \frac{1}{(1-\lambda)^2} \right\} \quad \begin{array}{l} \text{对于不成功查找} \\ \text{对于成功查找} \end{array}$$

使用几个  $\lambda$  值计算这个表达式的值，得到图 23-1 中的结果。当  $\lambda$  增大时——即散列表渐满——这些查找的比较次数也增大。这个结果与我们最初的直觉是一致的。例如，当散列表半满时——

23.3

23.4

| $\lambda$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9  |
|-----------|-----|-----|-----|-----|------|
| 不成功查找     | 1.1 | 1.5 | 2.5 | 6.1 | 50.5 |
| 成功查找      | 1.1 | 1.2 | 1.5 | 2.2 | 5.5  |

图 23-1 对于给定的装填因子  $\lambda$  值，使用线性探查在散列表中进行查找时所需的平均比较次数

即  $\lambda$  是 0.5——不成功查找平均需要 2.5 次比较，而成功查找平均需要 1.5 次比较。当  $\lambda$  大于 0.5 时，随着  $\lambda$  的增大，不成功查找的比较次数比成功查找的比较次数增长的快得多。所以，当散列表中已占据了一多半的位置时，性能迅速下降。如果发生这种情况，你必须定义更大的散列表，我们在本章稍后“再散列”一节会讨论这个问题。

 **注：**随着装填因子  $\lambda$  的增大，线性探查的散列性能明显下降。为保持合理的效率，散列表应该不超过半满。即保持  $\lambda < 0.5$ 。

**23.5 二次探查和双散列。**二次探查引起的二级聚集不如使用线性探查时发生的基本聚集那样严重。这里，对于给定的查找键，查找探查序列时比较的平均数约为

$$\frac{1}{(1-\lambda)} \quad \text{对于不成功查找}$$

$$\frac{1}{\lambda} \log\left(\frac{1}{1-\lambda}\right) \quad \text{对于成功查找}$$

对于在线性探查中使用过的相同的  $\lambda$  值，上述表达式的计算结果列在图 23-2 中。注意，随着  $\lambda$  值的增大，不成功查找的比较次数，比成功查找的比较次数增长快得多。虽然，随着  $\lambda$  值的增大，性能的降低不如线性探查那么严重，但是仍然要让  $\lambda < 0.5$  以保持效率。

尽管双散列可以避免线性探查和二次探查中的聚集，不过其效率的评估也与二次探查是一样的。

 **注：**如果使用二次探查或双散列，散列表应该不多于半满。即  $\lambda$  应该小于 0.5。

| $\lambda$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9  |
|-----------|-----|-----|-----|-----|------|
| 不成功查找     | 1.1 | 1.4 | 2.0 | 3.3 | 10.0 |
| 成功查找      | 1.1 | 1.2 | 1.4 | 1.7 | 2.6  |

图 23-2 对于给定的装填因子  $\lambda$  值，使用二次探查或双散列在散列表中进行查找所需的平均比较次数

## 拉链法的代价

**23.6** 对于拉链法解决冲突机制，散列表中的每个元素可以含有一个指向结点链表的引用。包括空链表在内，这样的链表的个数等于散列表的大小。所以装填因子  $\lambda$  是字典项个数除以链表的个数。即  $\lambda$  是每个链表中字典项的平均个数。因为这个数是平均数，故我们预料有些链表中含有少于  $\lambda$  个项——或者甚至没有——而有些则含有大于  $\lambda$  个项。假定链表无序，且字典中的查找键是唯一的。

字典操作 `getValue`、`remove` 和 `add` 中的每一个都需要在查找键对应的链表中进行查找。与开放地址情形一样，分析这些查找的效率就足够了。我们还是用装填因子  $\lambda$  来表示在散列表中查找查找键时所需要的比较次数。

散列表的不成功查找有时会遇到一个空链表，所以操作是  $O(1)$  的，这是最优情况。但当链表无序时，在散列表中查找一个项不成功的话，平均情况下要检查  $\lambda$  个结点。相反，成功查找总发生在链表不空的时候。成功查找时，除了要查看散列下标处表元素不是 `null` 外，还要查看平均有  $\lambda$  个结点的一个链表，且要查看  $\lambda/2$  个结点后才找到。所以当使用拉链法时，查找过程中的平均比较次数约为

$$\lambda \quad \text{对于不成功查找}$$

**$1+\lambda/2$  对于成功查找**

使用几个 $\lambda$ 值计算这些表达式，我们得到图 23-3 中的结果。当 $\lambda$ 增大时——即散列表渐满时——这些查找的比较次数仅略有增大。 $\lambda$ 的典型上界是 1，而更小的值也不会明显地改善性能。注意到不寻常的结果：当 $\lambda < 2$  时成功查找比不成功查找的时间更长。

记住这些结果都是平均情况。最差情况下，所有的查找键都映射到相同的表位置。所以，所有的项都出现在同一个结点链表中。当项数为 $n$ 时，最差查找时间则为 $O(n)$ 。

 **注：**拉链法散列的平均性能当装填因子 $\lambda$ 增大时没有明显下降。为保持合理的效率，应该让 $\lambda < 1$ 。

| $\lambda$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 | 1.1 | 1.3 | 1.5 | 1.7 | 1.9 | 2.0 |
|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 不成功查找     | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 | 1.1 | 1.3 | 1.5 | 1.7 | 1.9 | 2.0 |
| 成功查找      | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 | 2.0 |

图 23-3 对于给定的装填因子 $\lambda$ 值，使用拉链法在散列表中进行查找时所需的平均比较次数

 **注：保持散列性能**

冲突和冲突的解决一般会导致装填因子 $\lambda$ 的增大，且字典操作的效率降低。为保持效率，应该限制 $\lambda$ 的大小如下：

$\lambda < 0.5$  对于开放地址法

$\lambda < 1.0$  对于拉链法

如果装填因子超过这个界限，则必须增大散列表的长度，这个内容在下节讨论。

## 再散列

前一节讨论了基于散列机制实现字典时，使用不同冲突解决方案的效率。正如你所看到的，为保证实现的高效率，不能让装填因子 $\lambda$ 太大。很容易计算 $\lambda$ 值，看看它是否超出前面这个注中所给出的对应冲突解决方案的上限。

当 $\lambda$ 到达限值时该怎么办？首先，可以重定作为散列表的数组大小，如第 2 章所描述的。一般地，将原数组倍增，但此时必须保证数组的大小是素数。扩大数组为素数大小，且至少是原来的两倍大，这并不太困难。

通常，当扩展数组时，下一步是将原数组中的内容拷贝到新数组的对应位置。但散列表中不这样做。因为你已经改变了散列表的大小 $n$ ，相应的函数 $c \% n$ 会得到与原散列表中不同的下标。例如，如果原散列表中含有 101 个元素，则函数 $c \% 101$ 将散列码 505 压缩到下标 0。新的散列表含有 211 个元素，因为 211 是大于 2 倍 101 的最小素数。但现在 $c \% 211$ 将 505 压缩到地址 83。你不能简单地将原散列表中下标 0 的内容拷贝到新表的下标 0 中。而且也不能将它拷贝到新表的下标 83 处，因为还必须考虑冲突。

创建一个有合适大小的新的更大的散列表后，可以使用字典方法 add 将原散列表中的每个项添加到新表中。方法使用新表的大小计算散列下标并处理冲突。扩大散列表并为其内容计算新的散列下标的过程称为再散列（rehashing）。你可以看到，增大散列表的长度比增大原数组的大小，需要更多的工作。再散列不应是经常做的事情。

**注：再散列**

当装填因子 $\lambda$ 变得太大时，重设散列表的大小。要计算散列表的新尺寸，首先倍增当前个数，然后将这个值增大到下一个素数。使用方法add将字典中的当前项添加到新的散列表中。



**学习问题3** 考虑长度为5的散列表。函数 $c \% 5$ 将其散列码为20、6、18和14的项分别放到下标0、1、3和4中。展示当采用线性探查解决冲突机制时，再散列对这个散列表的影响。



**注：动态散列 (dynamic hashing)** 允许散列表的长度增大或减小，而不需要再散列的代价。我们并不准备介绍的这项技术，对保存于外部文件中的数据库的管理环境特别有用。

## 冲突解决机制的比较

23.8

前几段中，你已经看到装填因子 $\lambda$ 是如何影响不同冲突解决方法中查找散列表的平均比较次数的。图23-4图示了不同冲突解决机制下的这个影响。当 $\lambda$ 小于0.5时，不管解决冲突的过程如何，成功查找的平均比较次数差不多是一样的。对于不成功查找，当 $\lambda$ 小于0.5时，三种开放地址机制的效率差不多相同。但拉链法在这种情形下效率更高。

当 $\lambda$ 超过0.5时，开放地址法的效率迅速下降，而线性探查的效率最低。另一方面，拉链法对 $\lambda$ 值一直到1都能保持高效率。事实上，图23-3中的数据表显示当 $\lambda$ 介于1和2之间时效率仅微降。

拉链法无疑是最快的方法。但拉链法可能比开放地址法需要更多的空间，因为散列表中的每个元素都可能指向一个结点链表。另一方面，散列表本身可能比使用开放地址机制时更小，由此 $\lambda$ 可能更大。所以空间不是采用何种冲突解决方案的决定因素。

如果所有这些冲突解决机制都使用相同大小的散列表，则开放地址法比拉链法更可能导致再散列。为减少再散列的可能性，开放地址策略应该使用一个大的散列表。

在开放地址机制中，双散列是一个好的选择。它比线性探查使用更少的比较。另外，它的探查序列可以到达整个散列表，而二次探查则不能。

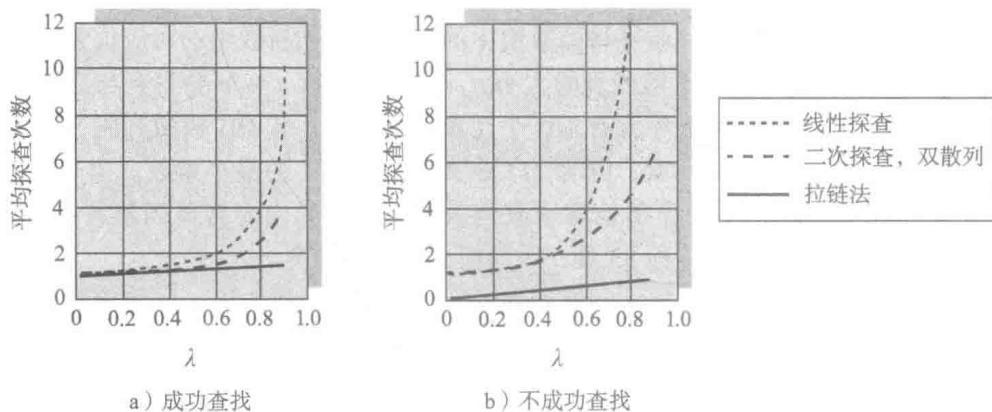


图23-4 4种冲突解决机制下，查找散列表所需的平均比较次数与 $\lambda$ 值的对比关系

## 使用散列实现字典

拉链法的效率使得它成为解决散列过程中所发生冲突的令人满意的方法。因为它的实现相对简单，所以将它留给读者来实现。我们来实现开放地址中的线性探查方法。字典实现中的大部分工作都不依赖于你使用的具体开放地址技术。只需很少的改动，就可以修改为使用二次探查或双散列来实现。

### 散列表中的项

散列表很像是第 21 章图 21-1a 中用来实现字典的数组。数组的每个元素都指向含有查找键及其对应值的一个对象。23.9

但是，对于开放地址法，散列表中的每个元素呈现 3 种状态之一：占用的、空的或是可用的。(见前一章段 22.16。) 空的位置含有值 null。我们可以修改散列表的结构让其表示其他的状态，或者让项对象含有这个表示。但可以使用一个更简单的办法。我们可以使用第 21 章程序清单 21-1 中给出的内部类 Entry 的实例作为散列表的项，将删除项替换为如下这样的其两个域都是 null 的项：

```
protected final Entry<K, V> AVAILABLE = new Entry<>(null, null);
```

所以将散列表中元素的状态区分如下：

- 空的——元素含有 null
- 可用的——元素指向 Entry 对象 AVAILABLE
- 占用的——元素指向字典中的一个项



### 设计决策：定义类 Entry

当我们第一次使用第 21 章中的 Entry 类时，将它定义为字典类的私有内部类。这是实现细节，很像在前面 ADT 的链式实现中用过的内部类 Node。在各种情形下，Node 和 Entry 类都是有用的类。到目前为止，我们简单地将内部类拷贝到需要它的任何其他类中。正如我们提到的，这些内部类是私有的。在我们将要讨论的类 HashedDictionary 中，将 Entry 声明为保护的。这样，HashedDictionary 的任何子类都可以访问 Entry。

如果 Node 和 Entry 这么有用，它们必须总是内部类吗？不是，但如果我们要让它们是顶层的公有类，它们将不再是实现细节。解决方案是，让这些类和在实现时使用这些类的类，像 HashedDictionary 类，在同一个包中。例如，第 3 章程序清单 3-5 展示了如何让类 Node 成为包的一部分。以类似的方式，可以如下这样写 Entry：

```
package DictionaryPackage;
class Entry<K, V>
{
 private K key;
 private V value;

 <此处是私有构造方法及私有方法 getKey, getValue 和 setValue。
 它们的定义在第 21 章程序清单 21-1 中。>
}
```

回忆一下，我们的字典不允许查找键是 null，也不允许其对应的值是 null。注意，Entry 的构造方法不做这个检查。虽然你可能认为它应该做，但如果它将构造时的任

何限制都留给客户去检查，则 Entry 将能用在更多的不同场景中。例如，我们刚定义的 Entry 对象 AVAILABLE 有 null 数据域。任何字典的实现可以保证，Entry 对象的查找键和值都不会是 null。事实上，在第 21 章，字典的 add 方法，作为唯一创建 Entry 对象的字典方法，做出这些保证。

## 数据域和构造方法

23.10

如果没有使用完美散列函数，则一定会有冲突。所有用于解决冲突的开放地址法都会随着散列表越来越满而降低效率，所以你必须增大散列表的尺寸。正如段 22.23 中提到的，增大表的大小能确保它含有 null 项——结束查找探查序列所必需的值。因为散列表是一个数组，故如段 23.7 中所述，可以扩展它并再散列各字典项。不过，我们修改装填因子  $\lambda$  的定义，将字典项的个数替换为占据或可用状态的表元素数。这个修改使得  $\lambda$  值变大了，所以在表中最后一个 null 项消失之前进行再散列。故类的开头列在程序清单 23-1 中。

**程序清单 23-1** 类 HashedDictionary 的框架

```

1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3 /**
4 A class that implements the ADT dictionary by using hashing and
5 linear probing to resolve collisions.
6 The dictionary is unsorted and has distinct search keys.
7 Search keys and associated values are not null.
8 */
9 public class HashedDictionary<K, V> implements DictionaryInterface<K, V>
10 {
11 // The dictionary:
12 private int numberEntries;
13 private static final int DEFAULT_CAPACITY = 5; // Must be prime
14 private static final int MAX_CAPACITY = 10000;
15
16 // The hash table:
17 private Entry<K, V>[] hashTable;
18 private int tableSize; // Must be prime
19 private static final int MAX_SIZE = 2 * MAX_CAPACITY;
20 private boolean integrityOK = false;
21 private static final double MAX_LOAD_FACTOR = 0.5; // Fraction of hash table
22 // that can be filled
23 protected final Entry<K, V> AVAILABLE = new Entry<>(null, null);
24
25 public HashedDictionary()
26 {
27 this(DEFAULT_CAPACITY); // Call next constructor
28 } // end default constructor
29
30 public HashedDictionary(int initialCapacity)
31 {
32 initialCapacity = checkCapacity(initialCapacity);
33 numberEntries = 0; // Dictionary is empty
34
35 // Set up hash table:
36 // Initial size of hash table is same as initialCapacity if it is prime;
37 // otherwise increase it until it is prime size
38 int tableSize = getNextPrime(initialCapacity);
39 checkSize(tableSize); // Check that size is not too large
40
41 // The cast is safe because the new array contains null entries

```

```

41 @SuppressWarnings("unchecked")
42 Entry<K, V>[] temp = (Entry<K, V>[])new Entry[tableSize];
43 hashTable = temp;
44 integrityOK = true;
45 } // end constructor
46
47 < Implementations of methods in DictionaryInterface are here.>
48 . .
49 < Implementations of private methods are here. >
50 . .
51 protected final class Entry<K, V>
52 {
53 < See Listing 21-1 in Chapter 21. >
54 } // end Entry
55 } // end HashedDictionary

```

域 `numberOfEntries` 记录字典中当前的项数。所以，当将项添加到字典中时，它的值增大，而删除项时，它的值减小。要区分字典的容量和散列表的长度，因为客户关心的是字典而不是它的实现。通过调用一个构造方法，客户可以创建一个字典，其初始容量可以是默认值或是用户选择的值。我们根据这个初始容量设定散列表的初始大小。但是，为了保证表的大小是素数，且至少大于所必需的值，第二个构造方法调用私有方法 `getNextPrime`，来找到大于等于所给整数的第一个素数。默认构造方法调用第二个构造方法，设定字典的默认初始容量。

**!** 程序设计技巧：为实现 `getNextPrime(anInteger)`，首先要看 `anInteger` 是不是偶数。如果是，它肯定不是素数，所以加上 1 让它成为奇数。然后在参数 `anInteger` 和其后的奇整数中间找第一个素数。

使用私有方法 `isPrime` 来测试一个整数是不是素数。为实现 `isPrime`，注意到 2 和 3 都是素数，但 1 及偶整数都不是。大于等于 5 的奇整数，如果它不能被最大到其平方根的每个奇整数整除，则是素数。

## 方法 `getValue`、`remove` 和 `add`

现在考虑字典的主要操作：`getValue`、`remove` 和 `add`。第 22 章段 22.17 结尾部分的“注”中简述了这些方法要完成的任务。

方法 `getValue`。我们从获取方法 `getValue` 的算法开始。

23.11

```

Algorithm getValue(key)
// Returns the value associated with the given search key, if it is in the dictionary.
// Otherwise, returns null.
if (找到key)
 return 找到项的值
else
 return null

```

则方法 `getValue` 的实现如下。

```

public V getValue(K key)
{
 checkIntegrity();
 V result = null;
 int index = getHashIndex(key);
 if ((hashTable[index] != null) && (hashTable[index] != AVAILABLE))
 result = hashTable[index].getValue(); // Key found; get value
}

```

```

 // Else key not found; return null
 return result;
} // end getValue

```

回忆一下，我们在第22章段22.22的结尾处，修改了私有方法getHashCode。

**23.12 方法remove。**类似于获取项，从散列表中删除一项也要涉及查找键的定位。如果找到，项的位置被标记为可用。下列伪代码描述了这个操作的必需步骤。

```

Algorithm remove(key)
// Removes a specific entry from the dictionary, given its search key.
// Returns either the value that was associated with the search key or null if no such object
// exists.
removedValue = null
index = getHashCode(key)
if (找到key) // hashTable[index] is not null and does not equal AVAILABLE
{
 removedValue = hashTable[index].getValue()
 hashTable[index] = AVAILABLE
 numberofEntries--
}
return removedValue

```

将remove的实现留作练习。

**23.13 方法add。**对于前面的方法getValue和remove，getHashCode方法内的探查步骤在散列表中查找给定的查找键。为此，它跳过含有null或AVAILABLE的位置。对于方法add，探查步骤执行类似的查找，但如果add中给定的查找键没有在散列表中，则add需要一个未占用位置的下标，用来在其中插入新项。如第22章段22.17说明的，这个位置应该尽可能地接近探查序列的开头，以加快后面对这个项的查找。下面给出的方法add的伪代码假定，由方法getHashCode来处理这些细节。

```

Algorithm add(key, value)
// Adds a new key-value entry to the dictionary. If key is already in the dictionary,
// it returns its corresponding value and replaces it in the dictionary with value.

if ((key == null) 或 (value == null))
 抛出一个异常
index = getHashCode(key)
if (没有找到key)
{ // Add entry to hash table
 hashTable[index] = new Entry(key, value)
 numberofEntries++
 oldValue = null
}
else // Search key is in table; replace and return entry's value
{
 oldValue = hashTable[index].getValue()
 hashTable[index].setValue(value)
}
// Ensure that hash table is large enough for another addition
if (散列表太满了)
 扩展散列表
return oldValue

```

这个算法暗示你要再写两个私有方法。我们将它们非正式地规范说明如下。

isHashTableTooFull()

如果散列表的装填因子大于等于MAX\_LOAD\_FACTOR，则返回真。这里我们将装填因子定义为locationsUsed与hashTable.length之比。

### enlargeHashTable()

扩展散列表，使得其长度是素数且至少两倍于原来的长度，然后将字典中的当前项添加到新的散列表中。为此，该方法必须再散列表中的各项。

将这些私有方法及公有方法 add 的实现留作练习。

**私有方法 enlargeHashTable。** 回忆一下，方法 `enlargeHashTable` 扩展散列表，使其表长是素数，且至少为原来的两倍大小。因为散列函数依赖于散列表的大小，所以不能将项从原数组中复制到新数组的相同位置。必须对每个项使用修改后的散列函数重新决定它在新表中应的位置。但这样做可能导致冲突，这是必须要解决的。所以应该使用方法 `add` 将已有的项添加到新的更大的散列表中。因为 `add` 增大了数据域 `numberOfEntries` 的值，所以必须记着在添加项之前将这个域置为 0。

方法的实现如下所示。

```
// Precondition: checkIntegrity has been called.
private void enlargeHashTable()
{
 Entry<K, V>[] oldTable = hashTable;
 int oldSize = hashTable.length;
 int newSize = getNextPrime(oldSize + oldSize);
 checkSize(newSize);

 // The cast is safe because the new array contains null entries
 @SuppressWarnings("unchecked")
 Entry<K, V>[] temp = (Entry<K, V>[])new Entry[newSize];
 hashTable = temp;
 numberOfEntries = 0; // Reset number of dictionary entries, since
 // it will be incremented by add during rehash

 // Rehash dictionary entries from old array to the new and bigger array;
 // skip elements that contain null or AVAILABLE
 for (int index = 0; index < oldSize; index++)
 {
 if ((oldTable[index] != null) && oldTable[index] != AVAILABLE)
 add(oldTable[index].getKey(), oldTable[index].getValue());
 } // end for
} // end enlargeHashTable
```

当遍历原来的散列表时，注意到，我们跳过了 `null` 元素和保存过已经从字典中删除的项的可用元素。

这个方法不再保留原散列表中的 `Entry` 实例。相反，它使用项的查找键和值创建新的项。你可以避免重新分配项；本章最后的练习 4 要求你研究这个可能性。



**学习问题 4** 当方法 `add` 调用 `enlargeHashTable` 时，`enlargeHashTable` 调用 `add`。但当 `enlargeHashTable` 调用 `add` 时，`add` 调用 `enlargeHashTable` 吗？请解释。

### 迭代器

最后，与第 21 章所做的一样，我们为字典提供迭代器。例如，我们可以实现一个内部类 `KeyIterator`，来定义查找键的迭代。这个迭代必须遍历散列表，忽略含有 `null` 或 `AVAILABLE` 的单元。图 23-5 所示为散列表的示例。深灰色的单元指向字典项，中度灰色的单元表示其中的项被删除后的可用位置，亮灰色的单元含有 `null`。当遍历这个表时，跳过亮灰色或中度灰色的单元。实现中唯一要关注的是检查迭代何时结束，即方法 `hasNext` 什

23.14

23.15

么时候应该返回假。不能依据单元的状态和散列表的大小进行这个判断。相反，你只需当方法 `next` 返回下一个查找键时，根据 `currentSize` 反着进行计数。

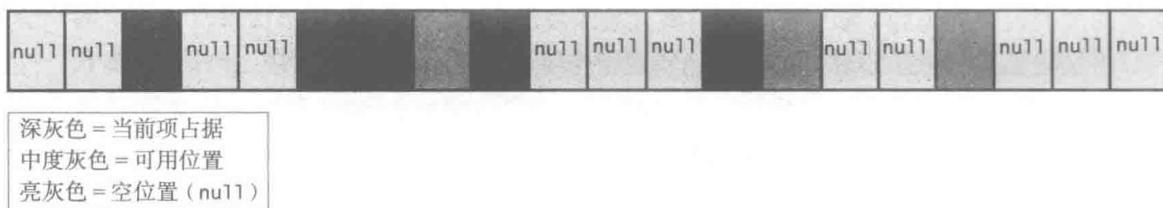


图 23-5 含有已占据的元素、可用元素和 null 值的散列表

`KeyIterator` 的实现如下所示。定义值迭代的类的实现是类似的。

```
private class KeyIterator implements Iterator<K>
{
 private int currentIndex; // Current position in hash table
 private int numberLeft; // Number of entries left in iteration

 private KeyIterator()
 {
 currentIndex = 0;
 numberLeft = numberOfEntries;
 } // end default constructor

 public boolean hasNext()
 {
 return numberLeft > 0;
 } // end hasNext

 public K next()
 {
 K result = null;
 if (hasNext())
 {
 // Skip table locations that do not contain a current entry
 while ((hashTable[currentIndex] == null) ||
 hashTable[currentIndex] == AVAILABLE)
 {
 currentIndex++;
 } // end while
 result = hashTable[currentIndex].getKey();
 numberLeft--;
 currentIndex++;
 }
 else
 throw new NoSuchElementException();
 return result;
 } // end next

 public void remove()
 {
 throw new UnsupportedOperationException();
 } // end remove
} // end KeyIterator
```



注：使用散列实现 ADT 字典时，没有提供对项进行排序的功能。这样的实现不适合用于要求对项进行有序迭代的应用中。

## Java 类库：类 `HashMap`

标准包 `java.util` 中含有类 `HashMap<K, V>`。这个类实现了我们在第 20 章段 20.22 中

提到的接口 `java.util.Map`。回忆一下，这个接口类似于我们的 `DictionaryInterface`。`HashMap` 假定，查找键对象属于重写了方法 `hashCode` 和 `equals` 的类。它的散列表是桶的集合，每个桶能含有若干项。正如你所知的，散列表的装填因子  $\lambda$  是衡量表满程度的量。`HashMap` 的构造方法能让你指定桶的初始值及最大装填因子  $\lambda_{\max}$ 。这些构造方法如下所示。

```
public HashMap()
```

使用默认的初始容量 16 和默认的最大装填因子 0.75 创建一个空映射（字典）。

```
public HashMap(int initialCapacity)
```

使用给定的初始容量和默认的最大装填因子 0.75 创建一个空映射（字典）。

```
public HashMap(int initialCapacity, float maxLoadFactor)
```

使用给定的初始容量和给定的最大装填因子创建一个空映射（字典）。

```
public HashMap(Map<? extends K, ? extends V> map)
```

使用 `map` 中的项创建一个映射（字典）。

`HashMap` 最初的设计者选择了默认最大装填因子 0.75 来折中时间和空间的需求。虽然装填因子越大，能让散列表越小，但会导致查找时间更长，反而会降低 `get`、`put` 和 `remove` 方法的效率。

当散列表中的项数超出  $\lambda_{\max}$  乘以桶数时，使用再散列增大散列表的大小。但再散列要花时间。如果选择

$$\text{桶数} > \frac{\text{字典中最大项数}}{\lambda_{\max}}$$

则可避免再散列。当然，散列表越大越浪费空间。

## Java 类库：类 `HashSet`

Java 类库中的包 `java.util` 中还含有类 `HashSet<T>`。这个类实现了我们在第 1 章段 1.22 中提到的接口 `java.util.Set`。23.17 回忆一下，`set` 是不含有重复项但又类似于包的集合。`HashSet` 使用前一段中介绍的类 `HashMap` 的实例，来保存集合中的项。

`HashSet` 中定义的构造方法如下所示。

```
public HashSet()
```

创建有默认初始容量 16 的空集合。`HashMap` 的底层实例使用的装填因子为 0.75。

```
public HashSet(int initialCapacity)
```

创建有给定初始容量的空集合。`HashMap` 的底层实例使用的装填因子为 0.75。

```
public HashSet(int initialCapacity, float loadFactor)
```

创建有给定初始容量的空集合。`HashMap` 的底层实例使用指定的装填因子。

## 本章小结

- 当字典大小与散列表大小的比值保持很小时，散列实现字典时的效率是高效的。这个比值称为装填因子。对于拉链法，装填因子应该小于 1，对于开放地址法，应该小

于 0.5。如果装填因子超出这些界限，你必须再散列散列表。

- 再散列是将散列表的大小扩大为一个素数且至少是原表的两倍的过程。因为散列函数依赖于散列表长，故不能简单地将项从原表中拷贝到新表中。相反，应该使用 add 方法将当前的所有项添加到新的散列表中。
- 与开放地址法相比，提供平均情况下更快的字典操作的拉链法，能使用更小的散列表，而再散列的频率更低。如果两种方法使用相同大小的数组用作散列表，则拉链法因为它的链表而使用了更多的内存。
- 用散列来实现字典时不能提供涉及有序查找键的操作。例如，不能简单地按序遍历关键字，找到给定范围内的键，或是找出最大或最小的查找键。
- 包 `java.util` 含有实现了接口 `Map<K, V>` 的类 `HashMap<K, V>` 和实现了接口 `Set<T>` 的类 `HashSet<T>`。

## 练习

1. 假定使用开放地址法解决冲突。现在假设散列表快要满了。为避免快满的散列表带来的坏性能，必须创建新的更大的散列表。
  - a. 将所有的项移到新散列表中需要哪些步骤？
  - b. 对散列函数会做什么处理？
2. 为确保探查的平均次数小于等于 4，使用下列冲突解决机制时，散列表具有的最大装填因子是多少？
  - a. 线性探查
  - b. 双散列
  - c. 拉链法
3. 修改段 23.13 给出的方法 `add` 的伪代码，让其允许字典中有重复的查找键。
4. 方法 `enlargeHashTable` 不保留原散列表中 `Entry` 的实例。如果方法 `add` 的参数是一个项而不是查找键和值，则它是能够保留的。写一个这样的附加的私有方法 `add`，然后修改 `enlargeHashTable`，让它保留原散列表中的 `Entry` 实例。
5. 假定名字集合由第 22 章练习 1 修改的类 `Name` 的实例所组成。对每个名字，假定用字符串表示一个昵称。且假定每个昵称是查找键，你打算将昵称 - 名字对添加到如段 23.16 所描述的作为类 `HashMap` 实例的字典中。
  - a. 假定计划将 1000 个项添加到这个字典中。创建不需要再散列而能容纳 1000 个项的类 `HashMap` 的实例。
  - b. 写出将 4 个昵称 - 名字对添加到字典中的语句。然后写出获取及显示对应于所选昵称的名字的语句。
6. 能使用散列表来实现优先队列吗？请解释。

## 项目

1. 完成程序清单 23-1 中开头的类 `HashedDictionary` 的实现。
2. 使用散列及拉链法实现 ADT 字典。每个桶使用一个结点链表。字典的项应该有唯一的查找键。
3. 重做项目 2，但每个桶使用 ADT 线性表而不是使用结点链表。线性表的哪种实现是合理的？
4. 实现第 22 章项目 3 设计的类 `PatientDataBase`。使用散列表来保存病人记录。编写程序论证并测试这个类。
5. 虽然散列表的两种实现可能有相同的平均比较次数，但它们的分布却是不同的。下面的实验将检验

线性探查和双散列的这种可能性。需要两个内容完全不同的名字列表：一个至少含有 1000 个名字，另一个至少含有 10 000 个名字。

a. 对线性探查和双散列这两种冲突解决机制，当散列表中保存 100 个对象时，确定不成功查找时平均比较次数为 1.5 次的装填因子。对这个装填因子，所需的散列表表长是多少？

b. 创建两个有合适大小的散列表，和两个对应的用来计数的空线性表。一个散列表使用线性探查，另一个使用双散列。对下列事情循环迭代 1000 次：

- 清空散列表。
- 从有 1000 个项的线性表中随机选择 100 个名字，插入散列表中。
- 从有 10 000 个项的线性表中随机选择 100 个名字，在散列表中查找每个名字。（每次查找都是不成功的，因为两个表没有共同的名字。）
- 统计 100 次查找时每个散列表的比较次数，在对应于散列表的线性表中记录下这个次数。

迭代完成后，每个线性表都将含有 1000 个值。每个值是查找 100 个名字所需的总比较次数。计算并显示每个线性表的平均值及标准差。希望每个散列表的平均比较次数等于 150（1.5 乘以 100）。

6. 修改前一个项目如下：

- 让用户输入想要的平均比较次数。
- 显示结果的柱状图。柱状图显示给定等长间隔下数据值的频率。使用最低的平均比较次数作为间隔长度。

7. 第 1 章将集合定义为不允许有重复项的包。为集合定义类，使用散列表保存集合中的项。

8.（游戏 / 数据库）考虑一个巨大的多人在线游戏（MMOG）所用的含有成千上万条记录的数据库。每条记录含有一个玩家的数据。当有人加入游戏中时，创建一个账号并指定一个唯一的记录号。这个号用来查找玩家记录。因为游戏太受欢迎，所以 MMOG 系统很少收到删除账号的请求。

当玩家的数量增加到过千时，开发人员意识到，他们不能将整个数据库都放在内存中。他们决定使用索引机制，即玩家的名字和记录号将保存在内存中，而仅当玩家注册到游戏时才导入整个记录。

使用线性表和散列式字典实现这个数据库系统。线性表应该将每个玩家保存为序言中项目 13 中描述的字符类的实例。每次创建新玩家账号时，将它添加到线性表尾，所以它的位置也是它的记录号。如果删除一个账号，它在线性表中的位置应该置为 null，所以其他项的位置不会改变。字典应该使用玩家的名字作为键。每个项的值是玩家的详细内容在线性表中的记录号（线性表位置）。你要做的实现决策之一是，数据库中使用的线性表类型。

9. 第 14 章讨论了计算 Fibonacci 数的低效的递归方法。效率低是因为有很多的方法调用都带有相同的参数。基于同样的原因，当递归计算阿克曼（Ackerman）函数时效率也低。这个函数的定义如下

$$A(m, n) = \begin{cases} n+1 & \text{如果 } m = 0 \\ A(m-1, 1) & \text{如果 } m > 0 \text{ 且 } n = 0 \\ A(m-1, A(m, n-1)) & \text{如果 } m > 0 \text{ 且 } n > 0 \end{cases}$$

这个函数相对于  $m$  和  $n$  的大小增长迅速，甚至当  $m$  和  $n$  很小时也需要递归调用很多次。

通过将前一次递归调用的结果保存在字典中的方法，可以加速阿克曼函数的计算，这样就不会重复那些调用了。设计并实现一个递归方法，对给定的  $m$  和  $n$  计算阿克曼函数。在计算  $A(m, n)$  之前，查找字典看看是否已经做过了。如果是，即如果  $A(m, n)$  在字典中，则返回它的值。如果没有，计算这个值，并将它保存在字典中，返回这个值。

你需要开发一个简单的使用  $m$  和  $n$  的散列函数。

10. 重做第 20 章项目 10，但使用程序清单 23-1 中概括的类 HashedDictionary，使用拉链法解决

冲突。开发一个算法，从一个给定的 `Person` 对象获取散列码  $c$ 。然后使用散列函数  $h(c) = c \bmod tableSize$ ，得到保存数据的位置。

11. 重做前一个项目，但这次

- a. 使用线性探查作为冲突解决机制
- b. 使用双散列作为冲突解决机制
- c. 使用二次探查作为冲突解决机制

12. 重做项目 10，但动态分配散列表。如果散列表已经半满了，则扩大它的大小到大于  $2 \times tableSize$  的第一个素数。

先修章节：第 5 章、第 9 章、第 12 章、Java 插曲 4、第 19 章

### 目标

学习完本章后，应该能够

- 使用标准术语描述二叉树和一般树
- 按前序、后序、中序或是层序遍历一棵树
- 给出二叉树示例，包括表达式树、决策树、二叉查找树和堆
- 给出一般树示例，包括解析树和游戏树

作为一种植物，树众所周知。作为组织数据的一种方式，你对树的熟悉程度超出了你的想象。家族树或是锦标赛参赛选手图是常见的树的两个例子。树提供一种层次结构，其中数据项有祖先和后代。该结构比你以前见到的任何结构都更丰富和多变。

本章研究 ADT 树的两种形式——二叉树和一般树——并展示几个应用这些树的示例。

## 树的概念

到目前为止，你见过的数据组织结构是按线性次序放置数据。栈、队列、线性表或是字典中的对象依次出现。与这样的组织方式同样有用的是，常常需要将数据分成组或是子组。这样的分类称为层次 (hierarchical) 或是非线性 (nonlinear)，因为数据项出现在结构的不同层中。24.1

我们先来看看几个熟悉的分层数据的示例。每个例子都使用代表一棵树的图来说明。

### 层次结构



24.2 示例：家族树。你的亲戚可以按多种方式组织为层次结构。图 24-1 是 Carole 的孩子和孙子们。她的儿子 Brett 有一个女儿 Susan。Carole 的女儿 Jennifer 有两个孩子，Jared 和 Jamie。

换一种分类方式，图 24-2 显示的是 Jared 的父母和祖父母。Jared 的父亲是 John，他的母亲是 Jennifer。John 的父母分别是 James 和 Mary；Jennifer 的父母分别是 Robert 和 Carole。

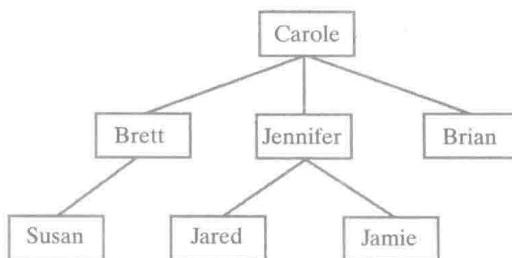


图 24-1 Carole 的孩子和孙子们

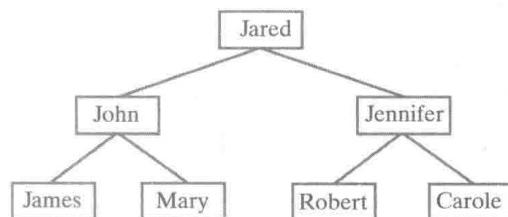


图 24-2 Jared 的父母和祖父母

24.3



**示例：大学的组织机构。**公司、学校、教派和政府都按层次来组织他们的人员。例如，图 24-3 显示一所典型大学行政机构的一部分。所有的办公室最终都向校长报告。校长下面有 3 位副校长。例如，负责教务的副校长监督各学院的院长。院长又监督各学术部系的系主任，例如计算机科学系与会计系。

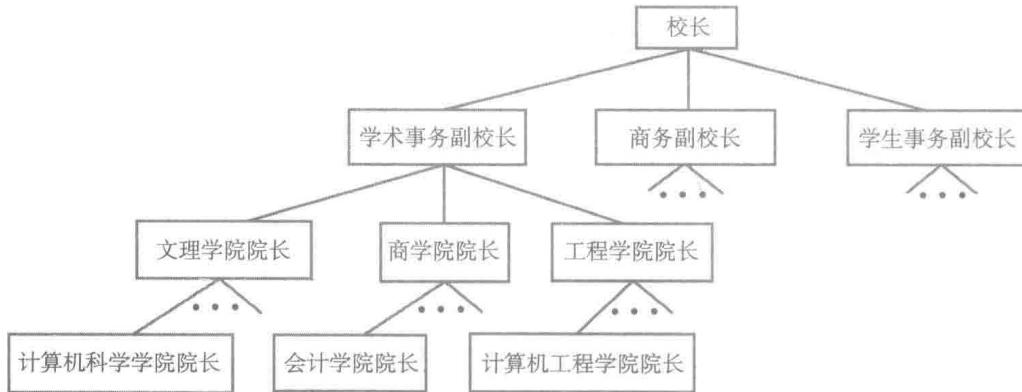


图 24-3 大学行政机构的一部分

24.4



**示例：文件目录。**一般来说，你将计算机内的文件组织到文件夹或目录中。每个文件夹又含有几个其他的文件夹或文件。图 24-4 显示的是 Paul 的计算机内的文件夹和文件的组织结构。该结构是层次的，即 Paul 的所有文件都组织在文件夹内，这些文件夹最终又放在文件夹 myStuff 中。例如，要找到预算文件，Paul 从文件夹 myStuff 开始，查找文件夹 home，最后找到文件 budget.txt。

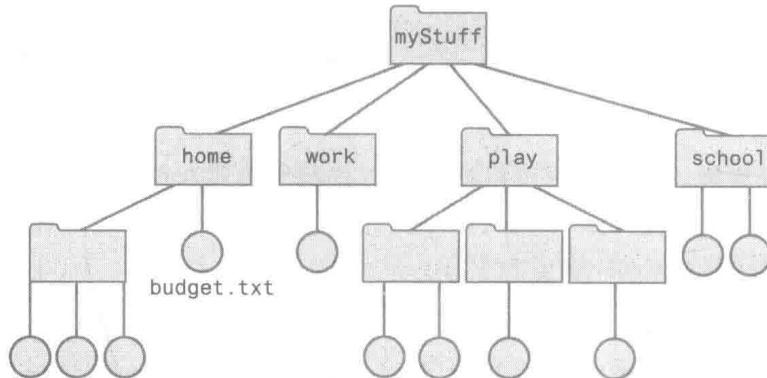


图 24-4 计算机文件被组织到文件夹中

## 树的术语

24.5

前面的每个图都是树的示例。树 (tree) 是一组由边 (edge) 相连的结点 (node)，边表示结点间的关系。结点按层 (level) 组织，层表示结点的层次。最上层的单结点称为根 (root)。图 24-5 显示了一棵树，除了结点名外，它与图 24-4 是一样的。在图 24-4 中，树根是文件夹 myStuff；在图 24-5 中，根是结点 A。

树中每个后继层中的结点是前一层中结点的孩子 (children)。有孩子的结点称为其孩子的父结点 (parent)。在图 24-5 中，结点 A 是结点 B、C、D 和 E 的父结点。因为这些孩子

有相同的父结点，故它们称为兄弟（ sibling）。它们也称为结点 A 的后代（ descendant），而结点 A 是它们的祖先（ ancestor）。此外，结点 P 是结点 A 的后代，而 A 是 P 的祖先。注意，结点 P 没有孩子。这样的结点称为叶子（ leaf）。不是叶结点的结点——即有孩子的结点——称为内部结点（ interior）或非叶结点（ nonleaf）。这样的结点也是父结点。

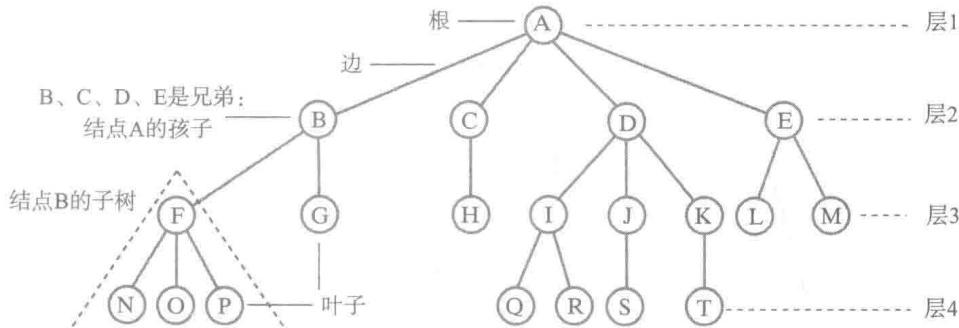


图 24-5 等价于图 24-4 中树的树

### 注：树

大多数植物的根牢牢地插在地下，而 ADT 树的根在树的最上面，它是层次结构的起点。每个结点都可以有孩子。有孩子的结点称为父结点；没有孩子的结点称为叶子。根是唯一没有父结点的结点；其他的所有结点，每个都有唯一的父结点。

### 学习问题 1 考虑图 24-5 中的树。

- a. 哪些结点是叶子？
- b. 哪些结点是结点 K 的兄弟？
- c. 哪些结点是结点 B 的孩子？
- d. 哪些结点是结点 B 的后代？
- e. 哪些结点是结点 N 的祖先？
- f. 哪些结点是父结点？

一般地，树中的每个结点可以有任意多个孩子。有时称这样的树为一般树（ general tree）。如果每个结点的孩子不多于  $n$  个，则树称为  $n$  叉树（  $n$ -ary tree）。要知道不是每棵一般树都是  $n$  叉树。如果每个结点最多有两个孩子，则树称为二叉树（ binary tree）。图 24-2 中的树是一棵二叉树，而前面其他几个图中的树是一般树。

24.6

### 注：树能否为空？

本书中任何的树都允许为空。有些书中只允许二叉树为空，但要求一般树至少含有一个结点。虽然这样要求的理由很合理，但我们不细究二叉树和一般树之间的这种细微差别，以避免混乱。

任意结点和它的后代组成原树的子树（ subtree）。结点的子树（ subtree of a node）是以该结点的孩子结点为根的一棵树。例如，图 24-5 中结点 B 的一棵子树是以 F 为根的树。树的子树（ subtree of a tree）是树根的子树。它以树根的孩子为根。



**学习问题2** 本书有可用树表示的层次结构。简述这棵树中的一部分，并用一般树或是二叉树来表示。

- 24.7 树的高度 (height) 是树中的层数。计算树的层数时，根从1层算起。图24-5中的树有4层，所以它的高度是4。单结点树的高度为1，空树的高度是0。

非空树的高度可以递归地通过其子树的高度来表示：

$$\text{树 } T \text{ 的高度} = 1 + T \text{ 的最高子树的高度}$$

图24-5中树的根有4棵子树，树高分别是3、2、3和2。因为这些子树最高为3，所以树高为4。

循着从根开始的沿着连接结点间的边从一个结点到另一个结点的路径 (path)，可以到达树中的任意结点。根和其他任一结点间的路径是唯一的。路径长度 (length of a path) 是组成路径的边数。例如，在图24-5中，经过结点A、B、F和N的路径长度为3。其他任何从根到叶子的路径都没有这条路径长。这棵树的高度是4，即为该最长路径长度加1。一般地，树的高度等于从根到叶子的最长路径的长度加1。换句话说，树的高度等于根与叶子之间最长路径上的结点个数。



注：树根和其他任何结点间的路径是唯一的。



注：树的高度等于树的层数。高度也等于根和叶子结点间最长路径上的结点数。



注：高度和层的另一种定义

有些书定义的树高和它的层数，都比本书给出的定义少1。例如，单结点树的高度是0而不是1。另外，树根在0层而不是在1层。



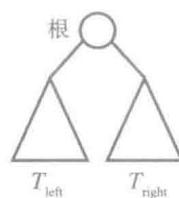
**学习问题3** 图24-1、图24-2和图24-3中的树的高度分别是多少？

- 24.8 二叉树。前面提到过，二叉树中的每个结点最多有两个孩子。它们称为左孩子 (left child) 和右孩子 (right child)。例如，图24-6中的每棵树都是一棵二叉树。图24-6a中，结点B、D和F都是左孩子，而结点C、E和G都是右孩子。该二叉树的根有两棵子树。左子树 (left subtree) 的根是B，而右子树 (right subtree) 的根是C。因此二叉树的左子树是其根的左子树；右子树也是如此。

二叉树中的每棵子树还是二叉树。实际上，可以递归地定义二叉树，如下所示。



注：一棵二叉树或者为空，或者有下列的形式：



其中  $T_{\text{left}}$  和  $T_{\text{right}}$  都是二叉树。

24.9

满二叉树和完全二叉树。高度为  $h$  的二叉树中，若其所有的叶结点都在  $h$  层上且每个非叶结点（父）都恰有两个孩子，则树称为满的（full）。图 24-6a 展示的是一棵满二叉树。如果二叉树中除最后一层外的所有层都含有最多的结点，最后一层的结点从左至右填充——如图 24-6b 所示——则树是完全的（complete）。图 24-6c 中的二叉树既不是满的也不是完全的。这样的树中，一个结点可以有左孩子但没有右孩子（例如结点 S），也可以有右孩子但没有左孩子（例如结点 U）。

 注：满二叉树中的所有叶结点都在同一层中，且每个非叶结点都恰有两个孩子。完全二叉树中，到倒数第二层都是满的，且最后一层的叶结点从左至右填充。二叉树用途广泛，这些特殊的树在后面的讨论中很重要。

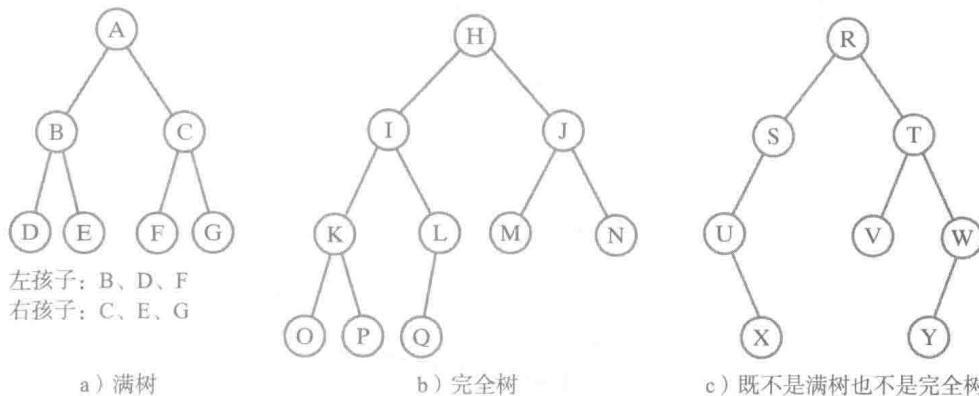


图 24-6 3 棵二叉树

平衡二叉树。若二叉树中每个结点有两棵高度完全相等的子树，则树称为完全平衡树（completely balanced）。唯一的完全平衡二叉树是满树。例如，图 24-6a 中的满树是完全平衡树。如果树中每个结点的子树的高度差不大于 1，则树称为高度平衡的（height balanced），或简称为平衡的（balanced）。完全二叉树——如图 24-6b 中的树——是高度平衡树，但有些非完全树也是高度平衡树，如图 24-7 所示。此外，平衡的概念适用于所有的树，不仅仅是二叉树。

24.10

二叉树中其子树的高度差不大于 1 的结点称为平衡结点（balanced node）。所以，平衡二叉树中的所有结点都是平衡的。

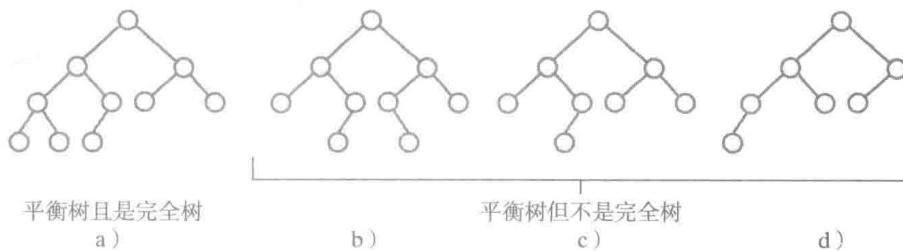


图 24-7 一些高度平衡二叉树示例

满树或完全树的高度。满树或是完全树的高度，在接下来几章有关效率问题的讨论中非常重要。图 24-8 所示为高度越来越高的几棵满树示例。每棵树中的结点数可表示为高度的函数。从图中每棵树的根开始，向叶结点方向移动时，每层的结点数倍增。图中最高的树中

24.11

结点总数是  $1+2+4+8+16$ , 即 31。一般地, 满二叉树的结点数是

$$\sum_{i=0}^{h-1} 2^i$$

其中  $h$  是树的高度。该累加和等于  $2^h - 1$ 。使用图 24-8 中的示例可以验证这个结论是正确的, 也可以作为练习用数学归纳法来证明。

如果满树的结点数为  $n$ , 则有下列结果:

$$n = 2^h - 1$$

$$2^h = n + 1$$

$$h = \log_2(n + 1)$$

即含  $n$  个结点的满树的高度是  $\log_2(n + 1)$ 。

含  $n$  个结点的完全树的高度是  $\log_2(n + 1)$  值向上舍入取整, 将该证明留作练习。

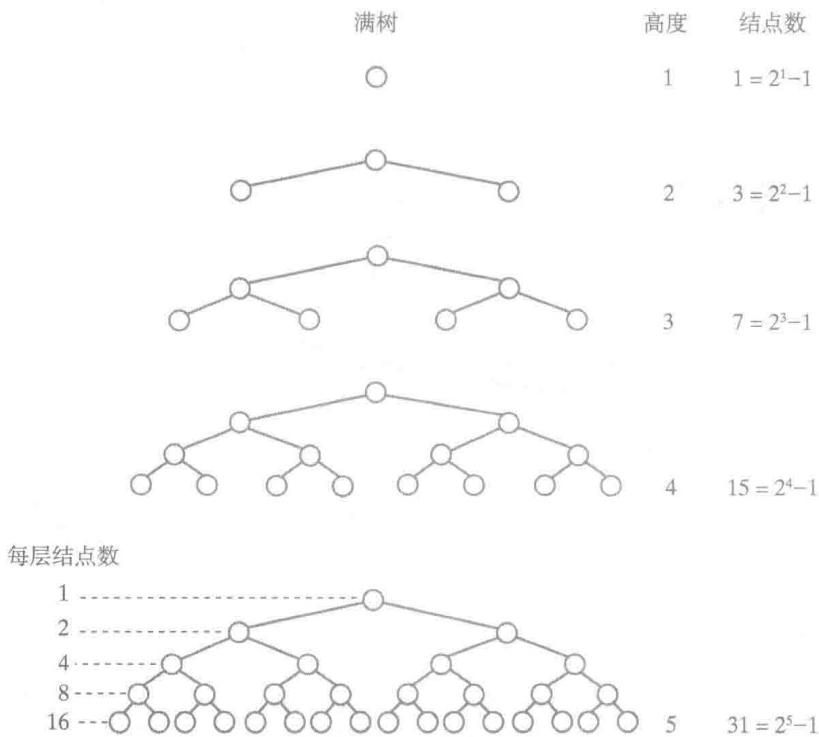


图 24-8 满二叉树中的结点数是树高的函数



注: 含  $n$  个结点的完全二叉树或是满二叉树的高度是  $\log_2(n + 1)$  向上取整。



**程序设计技巧:** 要在 Java 程序中计算  $\log_a x$ , 先观察  $\log_a x = \log_b x / \log_b a$ 。在 Java 中, `Math.log(x)` 返回  $x$  的自然对数。所以 `Math.log(x) / Math.log(2.0)` 得到以 2 为底  $x$  的对数。



**学习问题 4** 说明图 24-6a 和图 24-6b 中两棵二叉树的树高和树中结点数的关系是符合公式的。

**学习问题 5** 高度为 6 的满二叉树中有多少个结点?

**学习问题 6** 含有 14 个结点的完全树的高度是多少?

## 树的遍历

之前，树中结点的内容仅用作识别的标号。不过，因为树是 ADT，故其结点中包含着待处理的数据。现在来考虑包含了数据的结点。24.12

遍历数据集中的项是前几章见过的通用操作。那些例子中，数据按线性排列，所以遍历时项的次序很明确。树的情况与此不同。

在树的遍历或迭代中，对每个结点的数据项的处理只能恰好是一次。不过对项的访问次序不是唯一的。可以选择适合于具体应用的次序。因为二叉树的遍历比一般树的遍历更易理解，所以先介绍二叉树的遍历。为简单起见，使用术语“访问结点”表示“处理结点内的数据”的意思。

 注：“访问结点”表示“处理结点内的数据”的意思。这是树遍历过程中要执行的一个动作。遍历可以经过一个结点但在那个时刻并不访问它。要知道树的遍历是基于结点所处的位置，而不是基于结点中的数据值的。

## 二叉树的遍历

我们知道二叉树树根的子树还是二叉树。利用二叉树具有的递归特性来定义遍历是很自然的。要访问二叉树中的所有结点，必须24.13

访问根

访问根左子树中的所有结点

访问根右子树中的所有结点

在访问右子树中的结点之前先访问左子树中的结点，仅仅是个惯例，不过在每次遍历时必须保持一致。在访问两棵子树之前、中间或是之后访问根，由此定义了 3 种常见的遍历次序。第 4 种遍历次序用到了完全不同的方法。

在**前序遍历** (preorder traversal) 中，在访问根的子树之前访问根。然后访问根左子树中的所有结点，再访问根右子树中的所有结点。图 24-9 将二叉树的各结点按照前序遍历的访问次序标注了序号。先访问根，之后访问根左子树中的结点。因为这棵子树是二叉树，前序访问它的结点意味着在访问它的左子树之前访问它的根。按照这个递归方式继续遍历，直到访问完所有的结点。

**中序遍历** (inorder traversal) 访问二叉树根的子树中间访问二叉树的树根。一般地，按下列次序访问结点：

访问根左子树中的所有结点

访问根

访问根右子树中的所有结点

图 24-10 中将二叉树的各结点按照中序遍历的访问次序标注了序号。递归地访问左子树中的结点，导致首先访问最左叶结点。接下来访问那个叶结点的父结点，然后访问该父结点的右孩子。访问了根左子树中的全部结点之后访问树的根。最后，以这种递归方式访问根右子树中的结点。

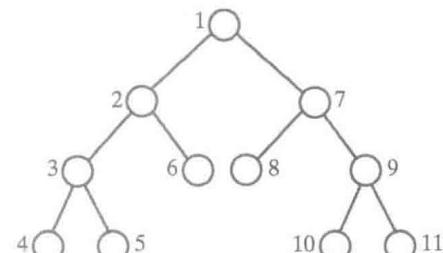


图 24-9 前序遍历的访问次序

24.14

24.15

后序遍历 (postorder traversal) 在访问了二叉树根的子树中的结点之后访问树的根。一般地，按下列次序访问结点：

访问根左子树中的所有结点

访问根右子树中的所有结点

访问根

图 24-11 中将二叉树的各结点按照后序遍历的访问次序标注了序号。递归地访问左子树中的结点，导致首先访问最左叶结点。然后访问叶结点的兄弟结点，然后是它们的父结点。访问了根左子树中的全部结点后，以这种递归方式访问根右子树中的结点。最后访问根。

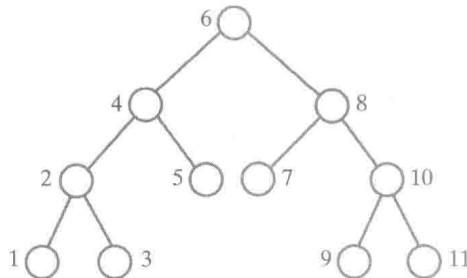


图 24-10 中序遍历的访问次序

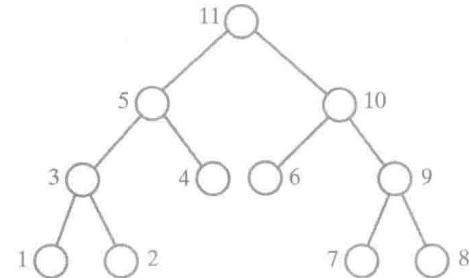


图 24-11 后序遍历的访问次序

24.16

层序遍历 (level-order traversal) —— 要讨论的最后一种遍历方法——从根开始，每次访问一层中的结点。同一层中，从左至右访问结点。图 24-12 将二叉树的各结点按照层序遍历的访问次序标注了序号。

层序遍历是广度优先遍历 (breadth-first traversal) 的示例。它访问的路径是，在移到下一层之前探查本层的全部结点。前序遍历是深度优先遍历 (depth-first traversal) 的示例。这种遍历全部探查完一棵子树之后再去探查另一棵子树。也就是说，遍历沿按树的层越来越深直到到达叶结点的路径进行。

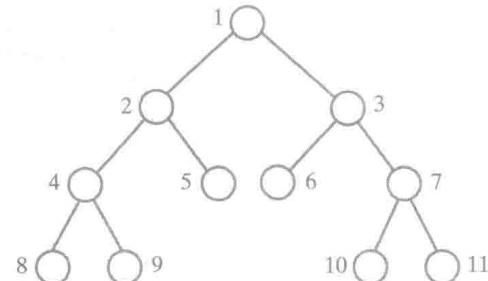


图 24-12 层序遍历的访问次序



### 注：二叉树的遍历

前序遍历在访问二叉树根的两棵子树结点之前，先访问树的根。

中序遍历在访问二叉树根的两棵子树结点的中间，访问树的根。

后序遍历在访问二叉树根的两棵子树结点之后，访问树的根。

层序遍历从根开始，在树的每层中从左至右访问结点。



**学习问题 7** 假定访问结点是指简单地显示结点中的数据。对图 24-2 所示的二叉树进行前序、后序、中序和层序遍历时，得到的结果分别是什么？

## 一般树的遍历

24.17

一般树的遍历有层序、前序和后序。对一般树而言，中序遍历的定义并不明确。

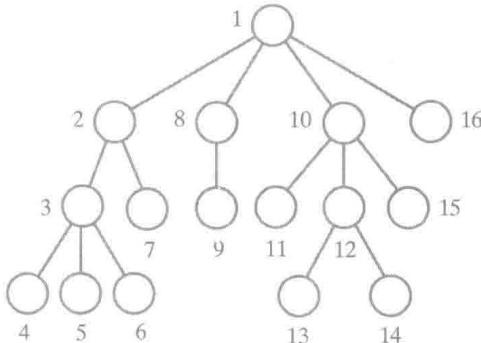
层序遍历一层层地访问结点，从树根开始。除了一般树中的每个结点可以有 2 个以上的

孩子结点之外，这个遍历就像是二叉树的层序遍历。

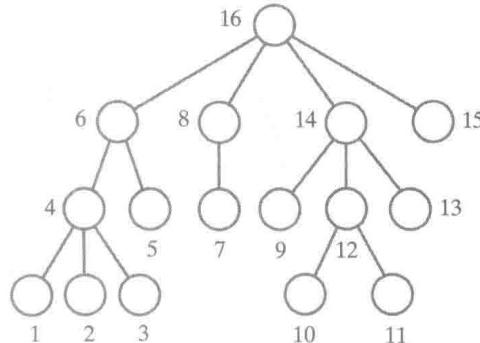
前序遍历访问根，然后访问根的每棵子树中的结点。后序遍历先访问根的每棵子树中的结点，最后访问根。图 24-13 给出一般树前序遍历和后序遍历的示例。



### 学习问题 8 层序遍历图 24-13 所示的树中结点，会得到什么次序？



a) 前序遍历



b) 后序遍历

图 24-13 一般树的两种遍历的访问次序

## 用于树的 Java 接口

树有许多不同树形和各种应用。为每种用途的 ADT 树写一个 Java 接口是件很笨的事情。相反，我们会整合具体应用的需求写若干个接口。将这些接口放到包里，同时还包括了实现它们的类。这样，包可以含有实现细节，例如结点的类，那正是我们想对树的客户隐藏的细节。第 25 章将介绍这些实现。

## 用于所有树的接口

**基本操作。**从规范说明所有的树通用操作的接口入手。程序清单 24-1 中的接口使用泛型 T 作为树结点中数据的类型。24.18

### 程序清单 24-1 所有树通用方法的接口

```

1 package TreePackage;
2 public interface TreeInterface<T>
3 {
4 public T getRootData();
5 public int getHeight();
6 public int getNumberOfNodes();
7 public boolean isEmpty();
8 public void clear();
9 } // end TreeInterface

```

该接口相当简单。它没有包括添加或删除结点的操作，哪怕这些操作的规范说明与这种树有关。这个接口中还不包括遍历操作，因为不是每种应用都用到遍历。相反我们为遍历提供一个单独的接口。

**遍历。**遍历树的一种方法是使用有 `hasNext` 和 `next` 方法的迭代器，这由 `java.util.Iterator` 接口提供。与前几章中的做法一样，可以定义一个方法返回一个这样的迭代器。24.19

因为有几种不同的遍历，所以树类中可以有多个方法，每个返回一种不同的迭代器。程序清单 24-2 为这些方法定义了一个接口。树类可以实现这个接口，根据需要的情况定义方法。

### 程序清单 24-2 树的遍历方法的接口

```

1 package TreePackage;
2 import java.util.Iterator;
3 public interface TreeIteratorInterface<T>
4 {
5 public Iterator<T> getPreorderIterator();
6 public Iterator<T> getPostorderIterator();
7 public Iterator<T> getInorderIterator();
8 public Iterator<T> getLevelOrderIterator();
9 } // end TreeIteratorInterface

```

### 用于二叉树的接口

**24.20** 实际上，树的很多应用中都用到了二叉树。可以将为一般树定义的 Java 类用于这些应用中，但使用专门为二叉树定义的类更方便有效。因为常常用到二叉树，所以为此专门开发 Java 的类还是值得的。

可以为基本的二叉树定义一个接口，将接口 `TreeInterface` 和 `TreeIteratorInterface` 中已有的方法添加进来。既然 Java 接口可以从多个接口派生，故为二叉树类写的接口列在程序清单 24-3 中。

### 程序清单 24-3 用于二叉树的接口

```

1 package TreePackage;
2 public interface BinaryTreeInterface<T> extends TreeInterface<T>,
3 TreeIteratorInterface<T>
4 {
5 /** Sets the data in the root of this binary tree.
6 * @param rootData The object that is the data for the tree's root.
7 */
8 public void setRootData(T rootData);
9
10 /** Sets this binary tree to a new binary tree.
11 * @param rootData The object that is the data for the new tree's root.
12 * @param leftTree The left subtree of the new tree.
13 * @param rightTree The right subtree of the new tree. */
14 public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
15 BinaryTreeInterface<T> rightTree);
16 } // end BinaryTreeInterface

```

`setTree` 方法将参数中所给的已有的二叉树对象，组合为一棵新树。它形成的树中，根结点含有给定的数据对象，两棵给定的二叉树是其子树。实现这个接口的类可能会有构造方法执行与这个方法相同的功能。但是，因为接口不能含有构造方法，所以没有办法强迫实现接口的类提供它们。

**24.21**  示例。假定类 `BinaryTree` 实现了接口 `BinaryTreeInterface`。要构造图 24-14 中的二叉树，先将每个叶结点表示为一棵单结点树。注意到树

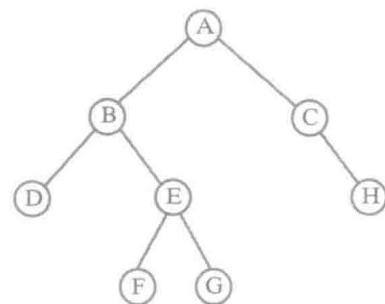


图 24-14 其结点含有单字符字符串的二叉树

中的每个结点都保存一个单字符的字符串。从叶结点开始向上，使用 `setTree` 建立越来越大的子树，直到得到整个的树。下面是建立树然后显示树的某些特征的 Java 语句。

```
// Represent each leaf as a one-node tree
BinaryTreeInterface<String> dTree = new BinaryTree<>();
dTree.setTree("D", null, null);
BinaryTreeInterface<String> fTree = new BinaryTree<>();
fTree.setTree("F", null, null);
BinaryTreeInterface<String> gTree = new BinaryTree<>();
gTree.setTree("G", null, null);
BinaryTreeInterface<String> hTree = new BinaryTree<>();
hTree.setTree("H", null, null);
BinaryTreeInterface<String> emptyTree = new BinaryTree<>();

// Form larger subtrees
BinaryTreeInterface<String> eTree = new BinaryTree<>();
eTree.setTree("E", fTree, gTree); // Subtree rooted at E
BinaryTreeInterface<String> bTree = new BinaryTree<>();
bTree.setTree("B", dTree, eTree); // Subtree rooted at B
BinaryTreeInterface<String> cTree = new BinaryTree<>();
cTree.setTree("C", emptyTree, hTree); // Subtree rooted at C
BinaryTreeInterface<String> aTree = new BinaryTree<>();
aTree.setTree("A", bTree, cTree); // Desired tree rooted at A

// Display root, height, number of nodes
System.out.println("Root of tree contains " + aTree.getRootData());
System.out.println("Height of tree is " + aTree.getHeight());
System.out.println("Tree has " + aTree.getNumberOfNodes() + " nodes");

// Display nodes in preorder
System.out.println("A preorder traversal visits nodes in this order:");
Iterator<String> preorder = aTree.getPreorderIterator();
while (preorder.hasNext())
 System.out.print(preorder.next() + " ");
System.out.println();
```



学习问题 9 前一个示例中展示的 Java 代码产生的输出是什么？

## 二叉树示例

现在看几个使用树来组织数据的示例，实现细节留待第 25 章讨论。第一个例子说明了本章前面介绍的一些遍历。

### 表达式树

可以用二叉树来表示其运算符为二元运算符的代数表达式。回忆第 5 章段 5.5，二元运算符有两个操作数。例如，表达式  $a/b$  可以表示为图 24-15a 所示的二叉树。树根含有运算符 /，根的孩子含有这个运算符的操作数。注意，孩子的次序要与操作数的次序相匹配。这样的二叉树称为表达式树 (expression tree)。图 24-15 中还有其他几个表达式树的例子。注意到表达式中的括号都不出现在树中。事实上，不需要括号，树也能得到表达式中运算符的次序。

24.22

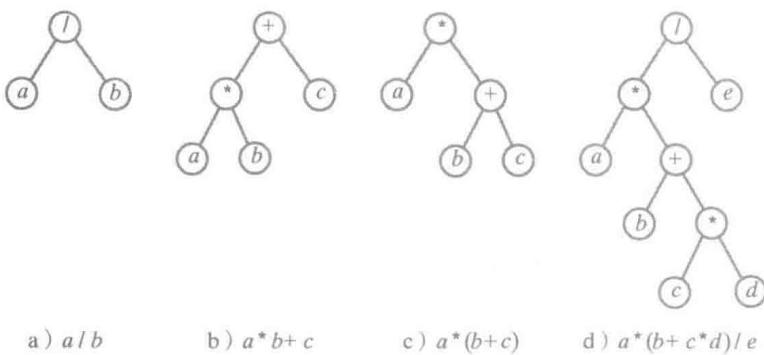


图 24-15 对应 4 个代数表达式的表达式树

**24.23** 段 5.5 提到, 代数表达式有不同的写法。正常书写的表达式, 即每个二元运算符出现在两个操作数的中间, 称为中缀表达式。前缀表达式将每个运算符放在其两个操作数的前面, 后缀表达式将每个运算符放在其两个操作数的后面。表达式树的不同遍历与表达式的这几种形式相关。

中序遍历表达式树, 按照它们在原中缀表达式中出现的次序访问树中的变量和操作数。如果访问每个结点时写出结点的内容, 则可得到中缀表达式, 但不包含括号。

前序遍历得到与原中缀表达式等价的前缀表达式。例如, 前序遍历图 24-15b 所示的树, 访问结点的次序是:  $+*a\ b\ c$ 。这个结果是中缀表达式  $a*b+c$  的前缀形式。回想一下, 与表达式树一样, 前缀表达式永远不包含括号。

后序遍历得到与原表达式等价的后缀表达式。后缀表达式也没有括号, 所以遍历可得到正确的结果。例如, 后序遍历图 24-15b 所示的树, 结点的访问次序为:  $a\ b*c+$ 。这个结果是中缀表达式  $a*b+c$  的后缀形式。



**学习问题 10** 对下列每个代数表达式, 写出对应的表达式树。

- a.  $a+b*c$
- b.  $(a+b)*c$

**学习问题 11** 前序、中序及后序遍历图 24-15a、图 24-15b、图 24-15c 和图 24-15d 中的树, 得到的结点访问次序分别是什么?

**学习问题 12** 图 24-15 中有满树吗? 有完全树吗? 有平衡树吗? 有的话, 分别指出。

**24.24**

**代数表达式的求值。**因为表达式树表示了表达式的运算次序, 故可以用它来计算表达式的值。表达式树的根总是一个运算符, 其操作数由根的左、右子树代表。如果我们能计算子树表示的子表达式的值, 则可以计算整个表达式的值。注意到, 如果我们知道了变量的值, 则图 24-15 中的每棵表达式树都可计算其值。

表达式树的后序遍历访问根的左子树, 然后是根的右子树, 最后访问根。如果访问子树时可以计算表达式的值, 那么将这些结果与根中的运算符一起可得到原表达式的值。故由下面递归算法可得到表达式树的值:

```
Algorithm evaluate(expressionTree)
if (expressionTree 为空)
 return 0
else
{
 firstOperand = evaluate(expressionTree 的左子树)
```

```

 secondOperand = evaluate(expressionTree的右子树)
 operator = expressionTree的根
 return 运算符operator作用于操作数firstOperand和secondOperand的结果
}

```

我们将在第 25 章实现表达式树。



**学习问题 13** 上述算法计算图 24-15b 所示的表达式树时返回什么值？假定  $a$  是 3， $b$  是 4， $c$  是 5。

24.25

## 决策树

 **示例：专家系统。**专家系统（expert system）帮助使用者解决问题或是做出决策。这样一个程序或许能帮助你选择专业或是申请奖学金。它根据你对一系列问题的回答得出结论。

决策树（decision tree）可作为专家系统的基础。决策树中每个父结点（非叶结点）是一个问题，它有有限个应答。例如，我们或许会使用其答案为 true 或 false、yes 或 no，或是有多种选择的问题。问题的每个可能答案都与该结点的一个孩子结点相对应。每个孩子结点可能是另一个问题，也可能是结论。作为结论的结点没有孩子结点，所以它们是叶结点。

一般来讲，决策树是  $n$  叉树，为的是它能放进多选问题。但通常决策树是一棵二叉树。例如，图 24-16 中的决策树是电视机故障诊断 yes-or-no 问题的二叉树的一部分。要使用这棵决策树，先显示根中的问题，根为当前结点（current node）。根据使用者的回答，转到相应的孩子结点——新的当前结点——并显示它的内容。所以，根据使用者的回答，我们沿着决策树中一条从根结点到叶结点的路径移动。在每个非叶结点显示一个问题。当到达叶结点时，提供结论。注意，二叉决策树中的每个结点或者有两个孩子结点，或者是叶结点。

决策树提供了能在树中的一条路径上移动及访问当前结点的操作。程序清单 24-4 包含了二叉决策树可能用到的 Java 接口。

### 程序清单 24-4 用于二叉决策树的接口

```

1 package TreePackage;
2 public interface DecisionTreeInterface<T> extends BinaryTreeInterface<T>
3 {
4 /** Gets the data in the current node.
5 * @return The data object in the current node, or
6 * null if the current node is null. */
7 public T getCurrentData();
8
9 /** Sets the data in the current node.
10 * @param newData The new data object. */
11 public void setCurrentData(T newData);
12
13 /** Sets the data in the children of the current node,

```

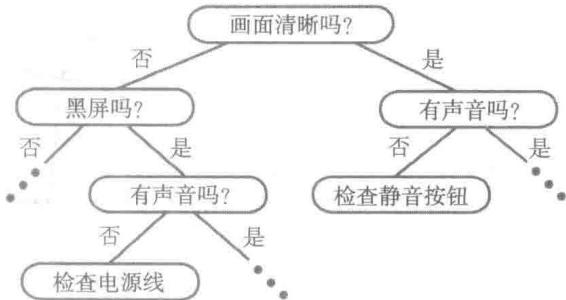


图 24-16 二叉决策树的一部分

```

14 creating them if they do not exist.
15 @param responseForNo The new data object for the left child.
16 @param responseForYes The new data object for the right child. */
17 public void setResponses(T responseForNo, T responseForYes);

18
19 /** Sees whether the current node contains an answer.
20 @return True if the current node is a leaf, or
21 false if it is a nonleaf. */
22 public boolean isAnswer();

23
24 /** Sets the current node to its left child.
25 If the child does not exist, sets the current node to null. */
26 public void advanceToNo();

27
28 /** Sets the current node to its right child.
29 If the child does not exist, sets the current node to null. */
30 public void advanceToYes();

31
32 /** Sets the current node to the root of the tree.*/
33 public void resetCurrentNode();
34 } // end DecisionTreeInterface

```

24.26



**示例：猜猜看游戏。**在猜猜看游戏中，你脑中想着一个事情，然后我通过问你 yes-or-no 问题来猜它是什么。假定程序问我问题。程序中使用一棵二叉决策树，随着游戏的展开，树逐渐增大。在使用之前程序并不创建它，而是从使用者那里获取事实，然后将它们加到决策树中。所以程序在游戏中学习，随着时间的推移变得越来越精通。

为简化问题，我们限制你选择的东西。例如，假定你选择一个国家。程序从简单的三结点树开始，如图 24-17 所示。

根据这棵树，程序问树根中的问题，并根据对问题的回答做出两种猜测中的一个。下面是程序和使用者（使用者的回答使用粗体显示）之间可能的交互：

```

Is it in North America?
> yes
My guess is U.S.A. Am I right?
> yes
I win.
Play again?

```

程序猜对了；树保持不变。

24.27

**猜猜看游戏中树的扩展。**假定使用者想的是另一个。交互可能是这样的：

```

Is it in North America?
> no
My guess is Brazil. Am I right?
> no
I give up; what are you thinking of?
> England
Give me a question whose answer is yes for England and no for Brazil.
> Is it in Europe?
Play again?

```

根据新的信息，我们可以扩展树，如图 24-18 所示。用使用者提供的新问题替换包含错误答案的叶结点的内容——本例中是 Brazil。然后给叶结点加上两个孩子结点。一个孩子中包含原叶结点中的猜测（Brazil），另一个包含使用者的答案（England），这个用作新的猜测。

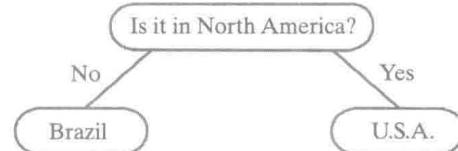


图 24-17 猜猜看游戏使用的初始决策树

现在，程序可以区分 Brazil 和 England 了。

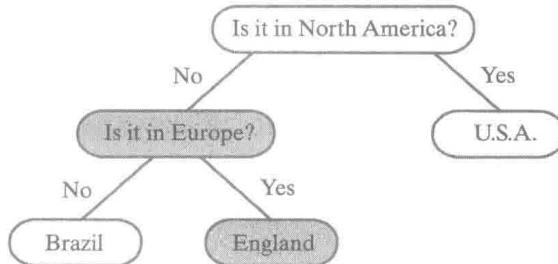


图 24-18 猜猜看游戏中，获取另一个事实后的决策树

## 二叉查找树

前几章讨论了数据查找的重要性。因为可以遍历任意树中的结点，所以肯定能在树中查找一个具体的数据。但是，用遍历去查找，可能与在数组中进行顺序查找一样效率不高。而用查找树（search tree）组织数据，可以让查找更有效率。本章，我们介绍最简单的一种查找树，即二叉查找树。第 28 章将介绍其他的查找树。

二叉查找树（binary search tree）是一棵二叉树，其结点含有 Comparable 类型的对象，并按下列规则组织：



注：对二叉查找树中的每个结点，

- 结点中的数据大于结点左子树中的所有数据
- 结点中的数据小于结点右子树中的所有数据

例如，图 24-19 所示为一棵名字的二叉查找树。作为字符串，Jared 大于 Jared 的左子树中的所有名字，但小于 Jared 的右子树中的所有名字。不仅仅是根满足这些特性，树中的每个结点都满足这些特性。注意到，Jared 的每棵子树本身也是二叉查找树。

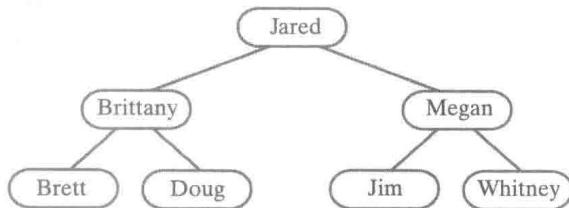


图 24-19 名字的二叉查找树



注：二叉查找树中的每个结点都是一棵二叉查找树的根。

前面对二叉查找树的定义，隐含表明树中所有的项都是不同的。强加这个限制是为了讨论更简单一些，不过可以修改定义，允许树中有重复的值。第 26 章讨论这种可能性。

二叉查找树的形态是不唯一的。即对同样的一组数据，可以组成几棵不同的二叉查找树。例如，图 24-20 显示了两棵与图 24-19 所示的树有相同名字的二叉查找树；可能还有其他的二叉查找树。



学习问题 14 由字符串 *a*、*b* 和 *c*，能组成多少棵不同的二叉查找树？

学习问题 15 在学习问题 14 得到的树中，最低和最高的树的高度分别是多少？

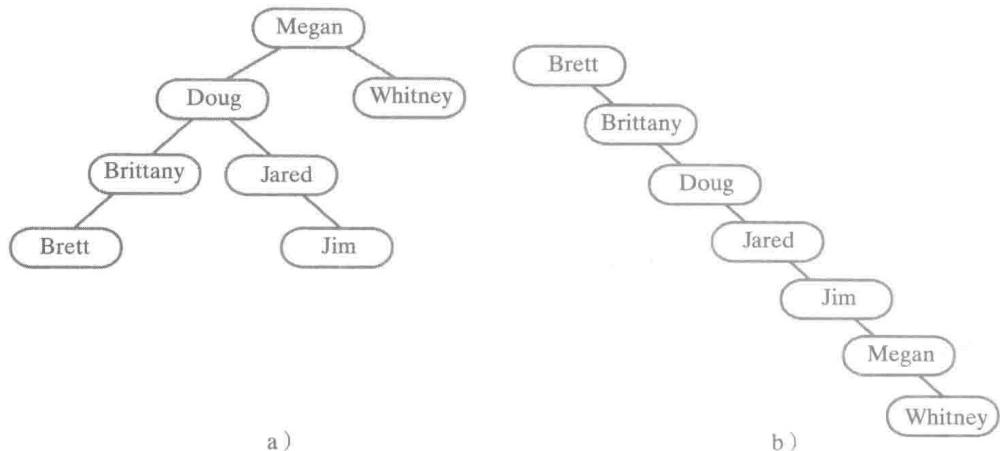


图 24-20 与图 24-19 所示的树有相同数据的两棵二叉查找树

24.30

**二叉查找树中的查找。**给定具体对象的查找关键字，二叉查找树中结点的组织方式能让我们在树中查找这个数据对象。例如，假定在图 24-19 所示的树中查找字符串 Jim。从树根开始，将 Jim 与 Jared 进行比较。因为字符串 Jim 大于字符串 Jared，所以在根的右子树中查找。将 Jim 和 Megan 进行比较，发现 Jim 小于 Megan。下一次在 Megan 的左子树中查找并找到 Jim。

要查找 Laura，将 Laura 和 Jared 进行比较，然后与 Megan 进行比较，再然后与 Jim 进行比较。因为 Laura 大于 Jim，应该在 Jim 的右子树中查找。但这棵子树是空树，则得到 Laura 不在树中的结论。

可以递归地描述查找算法：要在二叉查找树中进行查找，则在其一棵子树中进行查找。当找到要查找的项或是遇到一棵空子树时，查找结束。将这个查找过程形式化为下列伪代码：

```

Algorithm bstSearch(binarySearchTree, desiredObject)
// Searches a binary search tree for a given object.
// Returns true if the object is found.
if (binarySearchTree 为空)
 return false
else if (desiredObject == binarySearchTree 根中的对象)
 return true
else if (desiredObject < binarySearchTree 根中的对象)
 return bstSearch(binarySearchTree 的左子树, desiredObject)
else
 return bstSearch(binarySearchTree 的右子树, desiredObject)

```

这个算法有点像数组的二分查找。该算法是查找两棵子树中的一棵；二分查找是查找数组的一半。第 26 章可以看到该算法是如何实现的。

如果你认为使用二叉查找树能够实现 ADT 字典，那么你的回答是正确的。第 26 章将介绍这个是如何实现的。

24.31

**查找效率。**算法 bstSearch 沿二叉查找树的一条路径检查结点，从树根开始。当结点中含有所需的目标，或者结点是叶结点时，路径结束于此。前一段中，在图 24-19 所示的树中查找 Jim，检查了包括 Jared、Megan 和 Jim 在内的 3 个结点。一般地，成功查找所需的比较次数，等于路径中从根到含有所需项的结点间的结点个数。

在图 24-20a 中查找 Jim 需要 4 次比较；在图 24-20b 中查找 Jim 需要 5 次比较。图

24-20 中的两棵树都高于图 24-19 中的树。正如你所看到的，树高直接影响从根到叶的最长路径的长度，所以影响最坏情况下查找的效率。故在高度为  $h$  的二叉查找树中的查找效率是  $O(h)$  的。

注意，图 24-20b 中的树达到了 7 个结点的树能达到的最高高度。在这棵树上的查找，与在有序数组或是有序链表上的顺序查找，具有相同的性能。这些查找的效率都是  $O(n)$  的。

为使二叉查找树上的查找效率尽可能高，树必须尽可能低。图 24-19 中的树是满树，这是使用这些数据能够得到的最低的二叉查找树。在第 26 章将会看到，插入或删除结点会改变二叉查找树的形状。所以这样的操作可能会降低查找效率。第 28 章将介绍保持查找效率的策略。

## 堆

**定义。**堆（heap）是其结点含有 Comparable 类型对象的一棵完全二叉树，且满足以下条件。每个结点含有的对象不小于（或不大于）其后代结点中含有的对象。在最大堆（maxheap）中，结点中的对象大于等于其后代的对象。在最小堆（minheap）中，关系是小于等于。图 24-21 中给出了最大堆和最小堆的示例。为简单起见，图中使用整数而不是对象。

最大堆的根含有堆中最大的对象。注意，最大堆中任何结点的子树仍是最堆。尽管我们讨论的是最大堆，最小堆也有类似的模式。

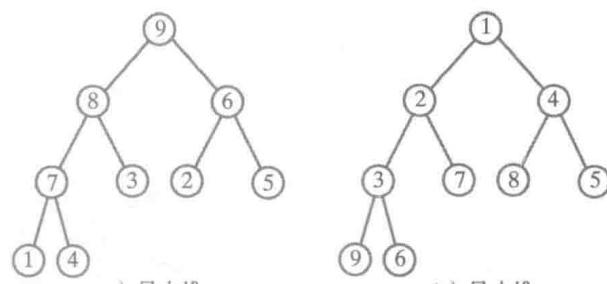


图 24-21 含有同样值的两个堆

**注：**最大堆是一棵完全二叉树，树中的每个结点含有一个 Comparable 对象，且大于等于该结点后代中的对象。与二叉查找树不同，堆中结点的子树之间没有关系。

**操作。**除了像 add、isEmpty、getSize 和 clear 这样的典型 ADT 操作外，堆还有获取和删除根中对象的操作。根据是最大堆还是最小堆，根中的对象或者是堆中的最大对象，或者是最小对象。可以利用堆的这个特性来实现 ADT 优先队列，下一段将会讨论。

程序清单 24-5 中的 Java 接口规范说明了最大堆的操作。

### 程序清单 24-5 用于最大堆的接口

```

1 public interface MaxHeapInterface<T extends Comparable<? super T>>
2 {
3 /** Adds a new entry to this heap.
4 @param newEntry An object to be added. */
5 public void add(T newEntry);
6
7 /** Removes and returns the largest item in this heap.
8 @return Either the largest object in the heap or,
9 if the heap is empty before the operation, null. */
10 public T removeMax();
11
12 /** Retrieves the largest item in this heap.
13 @return Either the largest object in the heap or,
14 if the heap is empty, null. */

```

```

15 public T getMax();
16
17 /** Detects whether this heap is empty.
18 * @return True if the heap is empty, or false otherwise. */
19 public boolean isEmpty();
20
21 /** Gets the size of this heap.
22 * @return The number of entries currently in the heap. */
23 public int getSize();
24
25 /** Removes all entries from this heap. */
26 public void clear();
27 } // end MaxHeapInterface

```

如果将项放在最大堆中，然后删除它们，则得到呈降序排列的各个项。所以可用堆来排序一个数组，第 27 章将讨论这个话题。



**学习问题 16** 含有一组给定对象的最大堆有唯一的根吗？用图 24-21a 中的最大堆作为示例来检验你的答案。

**学习问题 17** 含有一组给定对象的最大堆唯一吗？用图 24-21a 中的最大堆作为示例来检验你的答案。

#### 24.34

**优先队列。**可以使用堆来实现 ADT 优先队列。假定类 `MaxHeap` 实现了 `MaxHeapInterface`，实现了优先队列的类作为一个适配器类，开头的代码列在程序清单 24-6 中。回忆一下，我们在第 7 章段 7.19 中定义了 `PriorityQueueInterface`。

#### 程序清单 24-6 PriorityQueue 类的开头

```

1 public final class HeapPriorityQueue<T extends Comparable<? super T>>
2 implements PriorityQueueInterface<T>
3 {
4 private MaxHeapInterface<T> pq;
5
6 public HeapPriorityQueue()
7 {
8 pq = new MaxHeap<>();
9 } // end default constructor
10
11 public void add(T newEntry)
12 {
13 pq.add(newEntry);
14 } // end add
15
16 <Implementations of remove, peek, isEmpty, getSize, and clear are here.>
17 .
18 } // end HeapPriorityQueue

```

也可以让类 `MaxHeap` 实现 `PriorityQueueInterface`。然后定义字符串的优先队列，如下所示：

```
PriorityQueueInterface<String> pq = new MaxHeap<>();
```

### 一般树示例

本章的最后，给出一般树的两个示例。解析树在构造编译程序时很有用；游戏树是图 24-17 和图 24-18 介绍的决策树的推广。

## 解析树

第 14 章段 14.27 给出的如下规则，可用来描述字符串是否是有效的代数表达式：

- 一个代数表达式或者是一个项，或者是由运算符 + 或 - 分开的两个项。
- 一个项或者是一个因子，或者是由运算符 \* 或 / 分开的两个因子。
- 一个因子或者是一个变量，或者是一个包含在括号内的代数表达式。
- 变量是一个单字符。

第 14 章学习问题 13 要求你为由这 4 条规则定义的语言写一个语法。这个问题的答案如下。

```

<表达式> ::= <项> | <项> + <项> | <项> - <项>
<项> ::= <因子> | <因子> * <因子> | <因子> / <因子>
<因子> ::= <变量> | (<表达式>)
<变量> ::= a | b | ... | z | A | B... | Z

```

要看一个字符串是不是一个有效的代数表达式——即检查它的语法——我们必须要看是否能运用这些规则，从  $<\text{表达式}>$  派生出这个字符串。如果可以，则派生过程可表示为一棵解析树 (parse tree)，其中  $<\text{表达式}>$  是根，代数表达式的变量和运算符是其叶结点。表达式  $a^*(b + c)$  的解析树如图 24-22 所示。从树根开始，可知表达式是一个项。一个项又由两个因子得到。第一个因子是一个变量，具体来说就是  $a$ 。第二个因子是一个括在括号内的表达式。这个表达式是两个项的加法。每个项又各是一个因子；每个因子都是一个变量。第一个变量是  $b$ ；第二个是  $c$ 。因为能够形成这棵解析树，所以字符串  $a^*(b + c)$  是一个有效的代数表达式。

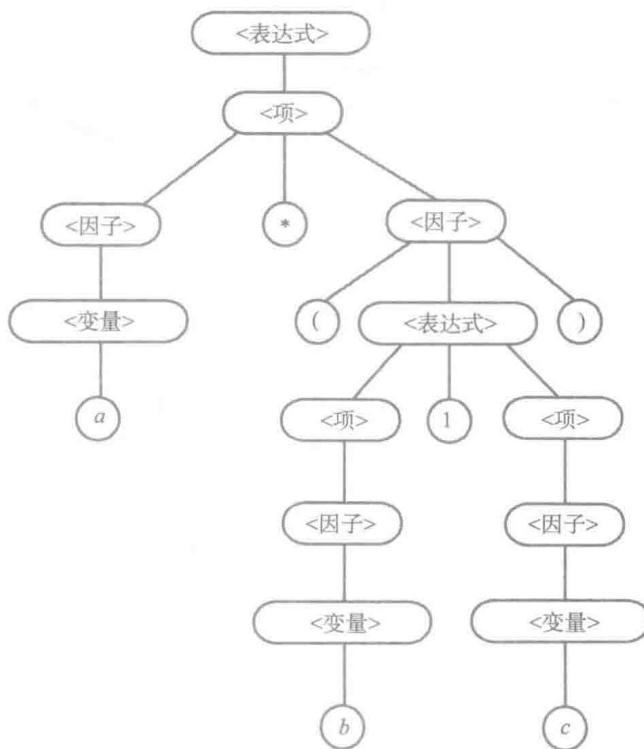


图 24-22 代数表达式  $a^*(b + c)$  的解析树



### 学习问题 18 画出代数表达式 $a*b + c$ 的解析树。

## 游戏树

**24.36** 对于像 tic-tac-toe (井字棋——译者注) 这样的双人游戏，可以用一般决策树来表示任何情况下可能的棋步。这样的决策树称为游戏树 (game tree)。如果树中一个给定结点表示一个玩家走完一步后的游戏状态，则该结点的孩子结点表示第二个玩家走一步棋后的可能状态。图 24-23 显示 tic-tac-toe 游戏树中的一部分。

可以在 tic-tac-toe 游戏程序中使用如图所示的游戏树。可以提前创建这棵树，也可以在游戏的过程中让程序来创建。不论哪种情形，程序都要确保树中不保留坏的棋步。这样，程序可以使用游戏树来改进棋艺。

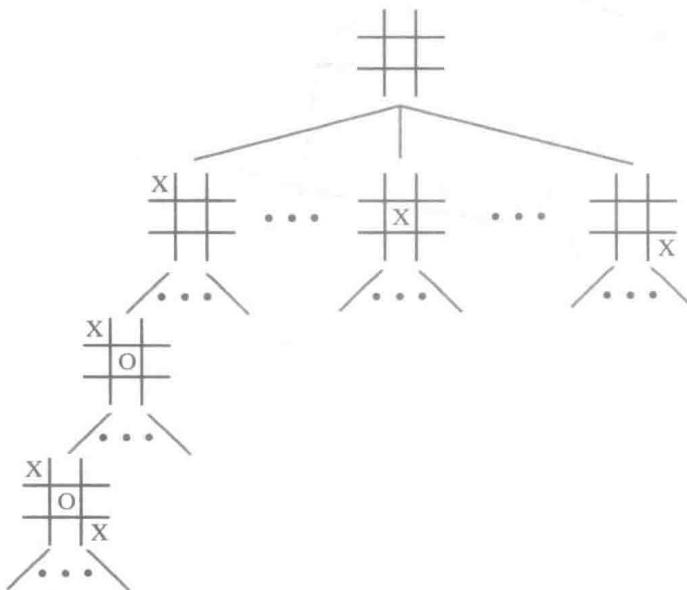


图 24-23 tic-tac-toe 的部分游戏树

## 本章小结

- 树是一组由边相连的结点，边表示结点间的关系。结点按层组织，层表示结点的层次。最上层的单结点称为根。
- 树中每个后继层中的结点是前一层中结点的孩子。没有孩子的结点称为叶结点。有孩子的结点是这些孩子的父结点。根是唯一没有父结点的结点。其他的所有结点，每个都有一个父结点。
- 二叉树中的结点最多有两个孩子。在  $n$  叉树中，一个结点最多有  $n$  个孩子。在一般树中，一个结点可以有任意多个孩子。

- 树的高度是树中的层数。高度还等于根与叶之间最长路径上的结点个数。
- 满二叉树中的所有叶结点都在同一层上，且每个非叶结点都恰有两个孩子。
- 高度为  $h$  的满树有  $2^h - 1$  个结点，这是能容纳的最多结点数。
- 完全二叉树直到倒数第二层都是满的。它最后一层的叶结点自左至右填充。
- 有  $n$  个结点的完全二叉树或满二叉树的高度是  $\log_2(n+1)$  值向上取整。
- 在完全平衡二叉树中，每个结点的子树都有完全相等的高度。这样的树必须是满的。如果树中每个结点的子树的高度差不大于 1，则树称为高度平衡树。
- 对每个结点恰好访问一次则遍历了树中的结点。几种遍历次序都是可能的。层序遍历从根开始，每次从左至右访问同一层中的结点。前序遍历中，在访问根的子树之前访问根。在后序遍历中，在访问根的子树之后访问根。对于二叉树，中序遍历先访问左子树中的结点，然后访问根，最后访问右子树中的结点。对于一般树，中序遍历没有明确定义。
- 表达式树是一棵二叉树，表示其运算符为二元运算符的代数表达式。表达式的操作数位于树的叶结点中。表达式中的任何括号都不在树中出现。可以使用一棵表达式树来计算代数表达式的值。
- 决策树在其每个非叶结点中都有一个问题。非叶结点的每个孩子都对应对问题的一种可能的响应。这些孩子结点中可以是另外一个问题，也可以是结论。作为结论的结点没有孩子结点，所以它们是叶结点。可以使用决策树来创建一个专家系统。
- 二叉查找树是一棵二叉树，其结点中含有 Comparable 类型的对象，并按下列规则组织：
  - ◆ 结点中的数据大于结点左子树中的数据
  - ◆ 结点中的数据小于结点右子树中的数据
- 二叉查找树中的查找快可达  $O(\log n)$ ，慢则为  $O(n)$ 。查找性能依赖于树形。
- 堆是其结点含有 Comparable 类型对象的一棵完全二叉树。每个结点中的数据不小于（或不大于）其后代中的数据。
- 可用堆实现优先队列。
- 某些规则形成描述代数表达式的语法。解析树是用来描述如何将这些规则用于具体表达式的一般树。可以使用解析树来检查所给表达式的语法。
- 游戏树是一棵一般决策树，其中含有某种游戏可能的棋步，例如像 tic-tac-toe 这样的游戏。

## 练习

1. 在第 14 章中，图 14-14a 显示了计算 Fibonacci 数列  $F_n$  的递归计算过程。回忆该数列的定义如下：

$$F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2} \quad \text{当 } n \geq 2$$

树根是  $F_6$  的值。 $F_6$  的孩子是计算  $F_6$  时必须有的两个值  $F_5$  和  $F_4$ 。注意到树的叶子含有基础情形  $F_0$  和  $F_1$  的值。

使用图 14-14a 作为示例，画出表示递归调用 mergeSort 的二叉树，该算法在第 16 章段 16.3 中给出。假定数组含有 20 个项。

2. 含有 21 个结点的二叉树高度最小是多少？满树呢？平衡树呢？

3. 考虑一棵 3 层的二叉树。

- a. 树中的结点个数最多是多少?
- b. 树中的叶结点最多是多少?
- c. 对一棵 10 层的二叉树, 回答前两个问题。
4. 编写计算二叉树中结点个数的递归算法。
5. 假定画一棵没有两个结点垂直对齐的二叉树。论证一条垂直线从左向右移动穿过树时, 所经结点的次序与中序遍历得到的结点次序是相同的。
6. 考虑二叉树的遍历。假定访问结点就是显示结点中的数据。对图 24-24a 所示的树分别进行下列遍历得到的结果是什么?
  - a. 前序遍历
  - b. 后序遍历
  - c. 中序遍历
  - d. 层序遍历

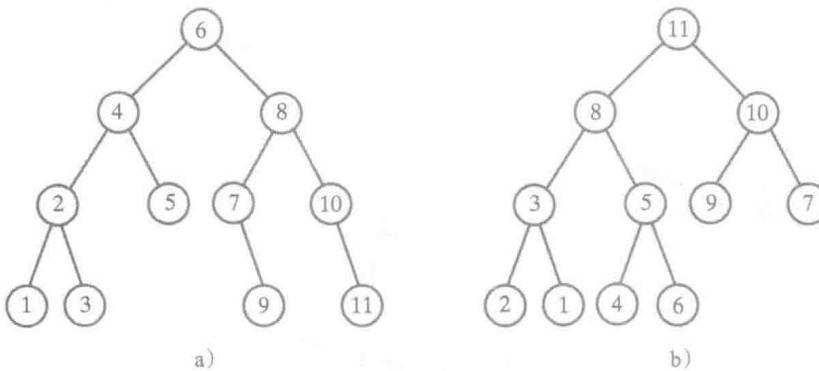


图 24-24 用于练习 6、练习 7 和练习 8 的两棵树

7. 使用图 24-24b 所示的树, 重做练习 6。
8. 图 24-24 中的两棵树含有整数。
  - a. 图 24-24a 中的树是二叉查找树吗? 为什么?
  - b. 图 24-24b 中的树是最大堆吗? 为什么?
9. 画出由以下字符串能够得到的最低的二叉查找树: Ann、Ben、Chad、Deepak、Ella、Jada、Jazmin、Kip、Luis、Pat、Rico、Scott、Tracy、Zak。你的树是唯一的吗?
10. 已知一棵二叉查找树的前序遍历结果是 6、2、1、4、3、7、10、9、11。树的后序遍历结果是什么?
11. 画出由练习 9 所给的字符串得到的最大堆。你的最大堆是唯一的吗?
12. 一棵二叉查找树可能是最大堆吗? 请解释。
13. 证明和

$$\sum_{i=0}^{h-1} 2^i$$

等于  $2^h - 1$ 。使用数学归纳法。

14. 二叉树的  $L$  层中最多有多少个结点? 使用数学归纳法证明你的答案。
15. 证明有  $n$  个结点的完全二叉树的高度是  $\log_2(n+1)$  值向上取整。
16. 假定按照层序遍历的访问次序为完全二叉树中的各结点进行编号。则树根编号为结点 1。图 24-25 所示为这样的一棵树。对结点  $i$ , 下列结点的编号是多少?
  - a. 兄弟结点, 如果有
  - b. 左孩子结点, 如果有

- c. 右孩子结点，如果有
- d. 父结点，如果有

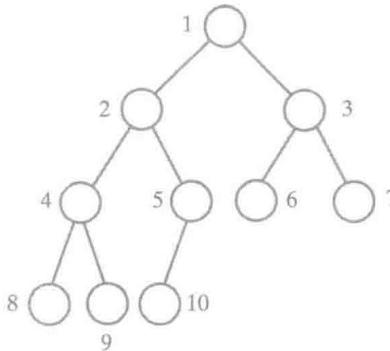


图 24-25 用层序遍历次序对完全二叉树的结点进行编号（练习 16）

17. 考虑一棵高度为  $h$  的满  $n$  叉树。其叶结点全部在最后一层。遍历这样一颗树的过程中,
  - a. 花在一个叶结点上的时间占比是多少？
  - b. 花在树最上面一半（从 1 层到  $h/2$  层）的结点上的时间占比是多少？
  - c. 对于  $n = 2, 10$  和  $100$ ，比较前两问中的时间占比。
18. 假定有  $n$  个值要加入一棵空二叉查找树中。
  - a. 将  $n$  个值添加到树中，有多少种不同的次序？这与含  $n$  个值的二叉查找树的个数不同。请解释为什么。
  - b. 图 24-20b 所示为一棵二叉查找树，其效率等同于有序表。将  $n$  个值添加到树中，使得树中每个父结点仅有一个孩子结点的不同次序有多少种？这样的树具有最坏的性能。
  - c. 随机构造具有最坏性能二叉查找树的概率是多少？提示：计算可能得到最差情况的总次数的占比。
19. 画出代数表达式  $(a + b)*(c-d)$  的表达式树。
20. 用段 24.24 所给的算法计算图 24-15c 所示的表达式树，返回值是多少？假定  $a$  是 3， $b$  是 4 且  $c$  是 5。
21. 画出下列每个代数表达式的解析树：
  - a.  $a + b*c$
  - b.  $(a + b)*(c-d)$
22. 为一般树开发接口 GeneralTreeInterface。

## 项目

1. 画出段 24.26 到段 24.27 叙述的猜猜看游戏的类图。  
对以下每个项目，假定你有一个类实现了段 24.20 中所给的 BinaryTreeInterface 接口。第 25 章将讨论这些实现细节。
2. 编写类似段 24.21 中的 Java 代码，创建一棵二叉树，其 8 个结点分别含有字符串 A、B、…、H，中序遍历树时以字典序访问结点。编写创建满树的一个版本，再写一个创建有最高高度的树的版本。中序遍历两棵树应该得到相同的结果。
3. 给定含 15 个任意次序的字符串的数组 wordList，编写 Java 代码，创建一棵满树，其中序遍历将返回字典序的字符串序列。提示：对字符串序列排序，然后用第 8 个串作为根。
4. 设计一个算法，由给定的后缀表达式产生一棵二叉表达式树。可以假定后缀表达式是一个仅含有二元运算符和单字符操作数的字符串。

5. 使用中缀表达式替代后缀表达式，重做前一个项目。

6. 给定类 `GeneralTree`，它实现了练习 22 中的接口 `GeneralTreeInterface`，实现程序，读入第 5 章项目 7 和项目 8 中提到的完全括号的 Lisp 表达式，创建表达式树。例如，表达式

```
(+ (- height)
 (* 3 3 4)
 (/ 3 width length)
 (* radius radius)
)
```

的表达式树如图 24-26 所示。

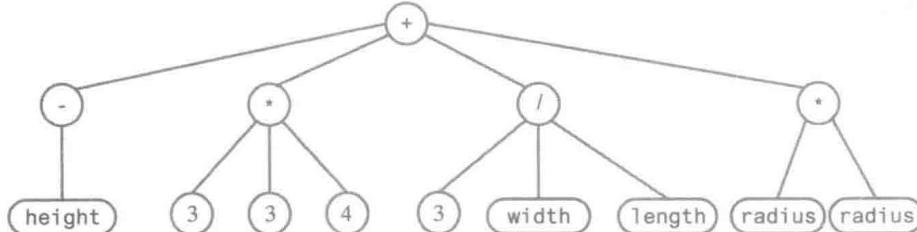


图 24-26 用于项目 6 的表达式树

7. 设计拼写检查器算法，至少含有下列方法：

- `void add(String word)`——将 word 添加到拼写检查器的拼写正确单词的集合中
- `boolean check(String word)`——如果所给的 word 拼写正确，则返回 true

在 26 叉树中保存拼写正确单词的集合。树中的每个结点有一个孩子，对应于字母表中的一个字母。每个结点还标示出从根到该结点路径所表示的字是否拼写正确。例如，图 24-27 所示的树中，给结点填充底色来做这样标记。这棵树中存储了单词“boa”、“boar”、“boat”、“board”、“hi”、“hip”、“hit”、“hop”、“hot”、“trek”和“tram”等。

要检查给定单词的拼写是否正确，可以从树根开始，沿着与单词的首字符对应的引用向下。如果引用为 `null`，则单词不在树中。否则，沿着与单词的第二个字符对应的引用向下，以此类推。如果最终到达一个结点，则检查它是否标记为拼写正确。例如，在图 24-27 所示的树中，“t”、“tr”和“tre”都是拼写错误的，而“trek”是拼写正确的。

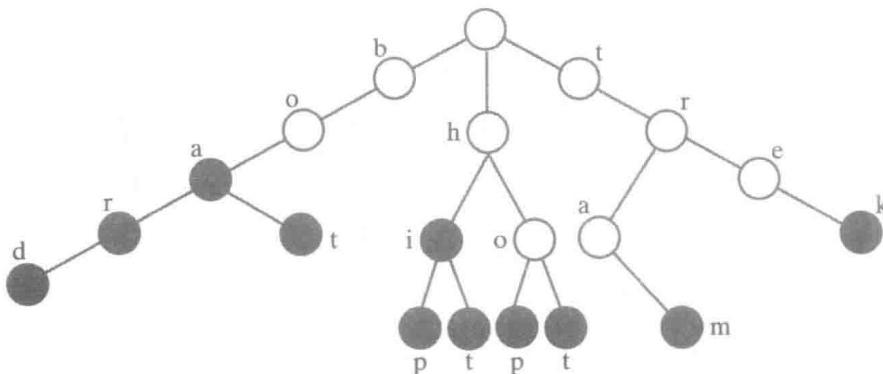


图 24-27 用于项目 7 的一般树

# 树的实现

**先修章节：**附录 C、Java 插曲 2、第 5 章、第 7 章、第 12 章、第 24 章

## 目标

学习完本章后，应该能够

- 描述对二叉树中结点的必要操作
- 实现二叉树中结点的类
- 实现二叉树的类
- 实现一棵表达式树
- 描述对一般树中结点的必要操作
- 使用二叉树来表示一般树

树最常见的实现是使用链式结构。结点，类似于用在链表中的结点，表示树中的每个元素。每个结点可以指向它的孩子，这是树中的其他结点。本章主要介绍二叉树，不过最后结束于对一般树的讨论。本章不涉及二叉查找树，后面用一整章来讨论它。

尽管可以使用数组或是向量来实现树，但本章不打算这样处理。这些实现方式仅对完全树才有吸引力。这样实现时，父子之间的链接不需要显式存储，所以其数据结构比不是完全树的要更简单。第 27 章将讨论完全树的一种应用，所以树的其他实现方式留待那时再讨论。

## 二叉树中的结点

树中的元素称为结点，与链表中的 Java 对象一样。我们使用类似的对象来表示树的结点，把它们也称为结点。25.1画在树中的结点，与表示它的 Java 结点之间的区别通常并不重要。

表示树中结点的结点对象指向数据域和结点的孩子。不管结点有多少个孩子，我们可以为所有的树定义结点类。但这样的类用于二叉树时不方便，效率也不高，因为二叉树的结点最多只有两个孩子。图 25-1 说明了二叉树的一个结点。它含有一个指向数据对象的引用和指向左孩子及右孩子的引用，左孩子及右孩子是树中的其他结点。指向孩子的两个引用都可以是 `null`。如果它们全是 `null`，则结点是叶结点。

虽然链表中的结点属于 `LinkedStack` 和 `LLList` 这样的类内的私有类 `Node`，但树结点的类并不是二叉树类内的类。因为派生于基础二叉树类的任何类，都可能需要对结点进行操作，故我们将树结点的类定义在二叉树类外。但我们并没有让结点类成为公有类。而是在包含不同的树类和它们的接口的包内，给它包访问权限。这样，结点的实现细节依然对树的客户隐藏。

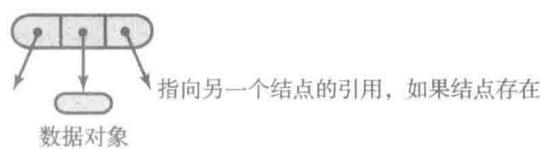


图 25-1 二叉树中的结点

注：链表中的结点对象指向链表中的另一个结点。虽然可以递归地处理链表，但结点并不指向一个链表。同样，二叉树中的结点对象也指向树中的其他结点。虽然我们常用递归思想看待二叉树，如第 24 章段 24.8 中那样，但树结点并不是指向另一棵树。

## 二叉结点类

25.2

程序清单 25-1 给出了用于二叉树的结点类的部分代码。我们将类放在 TreePackage 中，省略了它的访问修饰符。没有这个修饰符，类仅能被 TreePackage 包中的其他类访问。这些结点比链表中的结点要处理更多的事情。马上就会看到本类最后的 3 个方法是如何简化二叉树的实现的。

**程序清单 25-1** BinaryNode 类

```

1 package TreePackage;
2 class BinaryNode<T>
3 {
4 private T data;
5 private BinaryNode<T> leftChild;
6 private BinaryNode<T> rightChild;
7
8 public BinaryNode()
9 {
10 this(null); // Call next constructor
11 } // end default constructor
12
13 public BinaryNode(T dataPortion)
14 {
15 this(dataPortion, null, null); // Call next constructor
16 } // end constructor
17
18 public BinaryNode(T dataPortion, BinaryNode<T> newLeftChild,
19 BinaryNode<T> newRightChild)
20 {
21 data = dataPortion;
22 leftChild = newLeftChild;
23 rightChild = newRightChild;
24 } // end constructor
25
26 /** Retrieves the data portion of this node.
27 @return The object in the data portion of the node. */
28 public T getData()
29 {
30 return data;
31 } // end getData
32
33 /** Sets the data portion of this node.
34 @param newData The data object. */
35 public void setData(T newData)
36 {
37 data = newData;
38 } // end setData
39
40 /** Retrieves the left child of this node.
41 @return A reference to this node's left child. */
42 public BinaryNode<T> getLeftChild()
43 {
44 return leftChild;
45 } // end getLeftChild
46
47 /** Sets this node's left child to a given node.
48 @param newLeftChild A node that will be the left child. */
49 public void setLeftChild(BinaryNode<T> newLeftChild)
50 {
51 leftChild = newLeftChild;
52 } // end setLeftChild
53

```

```

54 /** Detects whether this node has a left child.
55 * @return True if the node has a left child. */
56 public boolean hasLeftChild()
57 {
58 return leftChild != null;
59 } // end hasLeftChild
60
61 <Implementations of getRightChild, setRightChild, and hasRightChild are
62 analogous to their left-child counterparts. >
63
64 /** Detects whether this node is a leaf.
65 * @return True if the node is a leaf. */
66 public boolean isLeaf()
67 {
68 return (leftChild == null) && (rightChild == null);
69 } // end isLeaf
70
71 /** Counts the nodes in the subtree rooted at this node.
72 * @return The number of nodes in the subtree rooted at this node. */
73 public int getNumberOfNodes()
74 {
75 <See Segment 25.10>
76 } // end getNumberOfNodes
77
78 /** Computes the height of the subtree rooted at this node.
79 * @return The height of the subtree rooted at this node. */
80 public int getHeight()
81 {
82 <See Segment 25.10>
83 } // end getHeight
84
85 /** Copies the subtree rooted at this node.
86 * @return The root of a copy of the subtree rooted at this node. */
87 public BinaryNode<T> copy()
88 {
89 <See Segment 25.5>
90 } // end copy
91 ...
92 } // end BinaryNode

```



注：一般地，表示树中结点的类，是要对客户隐藏的细节。省略它的访问修饰符，并将它放在实现树的类所在的包内，仅让包中的其他类访问它。

## ADT 二叉树的实现

第 24 章描述了几种不同的二叉树。例如，表达式树和决策树，每个都具有二叉树基本操作之外的一些操作。我们将二叉树类定义为像表达式树类这样的类的父类。

### 创建基本二叉树

第 24 章段 24.20 为二叉树类定义了下列接口：

25.3

```

public interface BinaryTreeInterface<T>
 extends TreeInterface<T>, TreeIteratorInterface<T>
{
 public void setRootData(T rootData);
 public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
 BinaryTreeInterface<T> rightTree);
} // end BinaryTreeInterface

```

回忆一下，段 24.18 中的 TreeInterface 规范说明了所有树共有的基本操作——getRootData、getHeight、getNumberOfNodes、isEmpty 和 clear，而段 24.19 中的 TreeIteratorInterface 规范说明了遍历树的操作。这 3 个接口都在我们的包 TreePackage 中。

实现二叉树先从构造方法和 setTree 方法入手，它们列在程序清单 25-2 中。私有方法 initializeTree 的形参类型是 BinaryTree，而接口中规范说明的公有方法 setTree 的形参类型是 BinaryTreeInterface。在实现 setTree 时调用了这个私有方法，为的是简化从 BinaryTreeInterface 到 BinaryTree 的转型。

第三个构造方法——其形参类型是 BinaryTree——也调用了 initializeTree。如果它调用的是 setTree，则应该将 setTree 声明为终极方法，这样就不能有子类来重写它，因此也就不会改变构造方法产生的效果了。还要注意，可以将私有方法命名为 setTree，而不是叫 initializeTree。

### 程序清单 25-2 BinaryTree 类的初稿

```

1 package TreePackage;
2 import java.util.Iterator;
3 import java.util.NoSuchElementException;
4 import StackAndQueuePackage.*; // Needed by tree iterators
5 /**
6 A class that implements the ADT binary tree.
7 */
8 public class BinaryTree<T> implements BinaryTreeInterface<T>
9 {
10 private BinaryNode<T> root;
11
12 public BinaryTree()
13 {
14 root = null;
15 } // end default constructor
16
17 public BinaryTree(T rootData)
18 {
19 root = new BinaryNode<T>(rootData);
20 } // end constructor
21
22 public BinaryTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree)
23 {
24 initializeTree(rootData, leftTree, rightTree);
25 } // end constructor
26
27 public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
28 BinaryTreeInterface<T> rightTree)
29 {
30 initializeTree(rootData, (BinaryTree<T>)leftTree,
31 (BinaryTree<T>)rightTree);
32 } // end setTree
33
34 private void initializeTree(T rootData, BinaryTree<T> leftTree,
35 BinaryTree<T> rightTree)
36 {
37 <FIRST DRAFT - See Segments 25.4 - 25.7 for improvements.>
38 root = new BinaryNode<T>(rootData);
39
40 if (leftTree != null)
41 root.setLeftChild(leftTree.root);
42 if (rightTree != null)
43 root.setRightChild(rightTree.root);

```

```

44 } // end initializeTree
45
46 < Implementations of setRootData, getRootData, getHeight, getNumberOfNodes,
47 isEmpty, clear, and the methods specified in TreeIteratorInterface are here. >
48 ...
49 } // end BinaryTree

```

**!** 程序设计技巧：当将 `BinaryTree` 类型的实例传给形参类型为 `BinaryTreeInterface` 的方法时，不需要进行转型。但是反过来是需要转型的。

## 方法 `initializeTree`

问题。上面给出的 `initializeTree` 的实现，不足以处理方法可能遇到的所有情况。假定客户定义了 `BinaryTree` 类的 3 个独立实例——`treeA`、`treeB` 和 `treeC`——并执行下列语句

```
treeA.setTree(a, treeB, treeC);
```

因为 `setTree` 调用了 `initializeTree`，所以 `treeA` 与 `treeB` 和 `treeC` 共享了结点，如图 25-2 所示。如果客户修改了树，比如是 `treeB`，则 `treeA` 也改变了。一般来讲这个结果是不受人欢迎的。

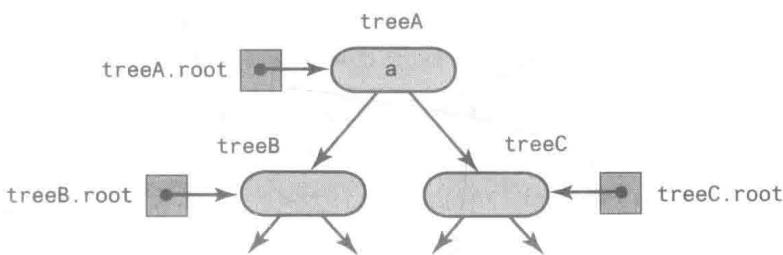


图 25-2 二叉树 `treeA` 与 `treeB` 和 `treeC` 共享结点

**?** 学习问题 1 客户改变 `treeB` 时导致 `treeA` 也改变了，我们说这是不受欢迎的。为什么说这种情况是危险的？

解决方案一。`initializeTree` 的一种解决方案是拷贝 `treeB` 和 `treeC` 中的结点。则 `treeA` 将与 `treeB` 和 `treeC` 分开。以后不论是对 `treeB` 还是对 `treeC` 的改变都不会影响到 `treeA`。我们来讨论这个方法。

因为拷贝了结点，所以用到了 `BinaryNode` 类中定义的 `copy` 方法。要拷贝一个结点，实际上必须拷贝以那个结点为根的子树。从那个结点开始，拷贝该结点，然后拷贝其左子树和右子树中的结点。所以执行了子树的前序遍历。为了简化操作，我们不拷贝结点中的数据。即一个结点和它的拷贝将含有相同的数据。

类 `BinaryNode` 中定义的 `copy` 方法如下：

```

public BinaryNode<T> copy()
{
 BinaryNode<T> newRoot = new BinaryNode<>(data);
 if (leftChild != null)
 newRoot.setLeftChild(leftChild.copy());
}

```

```

if (rightChild != null)
 newRoot.setRightChild(rightChild.copy());
return newRoot;
} // end copy

```

现在 `initializeTree` 可以调用 `copy` 方法来拷贝两棵给定子树中的结点了：

```

private void initializeTree(T rootData, BinaryTree<T> leftTree,
 BinaryTree<T> rightTree)
{
 root = new BinaryNode<>(rootData);
 if ((leftTree != null) && !leftTree.isEmpty())
 root.setLeftChild(leftTree.root.copy());
 if ((rightTree != null) && !rightTree.isEmpty())
 root.setRightChild(rightTree.root.copy());
} // end initializeTree

```

因为拷贝结点的开销大，所以我们考虑用另一种方案实现这个方法。下面会看到，我们只需在特定的情况下才需要拷贝一些结点。

**25.6** 另一种办法，引来更多的问题。不需要每次都拷贝结点，`initializeTree` 的动作可以如下所示。还是来看前面的例子，

```
treeA.setTree(a, treeB, treeC);
```

`initializeTree` 可以先将 `treeA` 的根结点与 `treeB` 和 `treeC` 的根结点链接起来。然后将 `treeB.root` 和 `treeC.root` 置为 `null`。这个方案解决了一个结点出现在多棵树中的问题，但它让客户作为参数传递的树变为了空树。结果，带来了另外两个困难。

假定客户执行语句

```
treeA.setTree(a, treeB, treeC);
```

如果 `initializeTree` 让子树 `treeA` 和 `treeB` 为空，则 `setTree` 将新的 `treeA` 也丢了！

如果客户执行如下的语句，则会出现另一个问题

```
treeA.setTree(a, treeB, treeB);
```

这种情况下，`treeA` 的根的左子树和右子树是同一棵了，如图 25-3 所示。这个问题的解决办法是拷贝 `treeB` 的结点，让子树分开。所以，一般情况下不能避免结点的拷贝，不过这样的拷贝不常发生。

下面的方案解决了这些问题。

**25.7** **解决方案二。** 概括来说，`initializeTree` 应该执行下列步骤：

- 1) 用给定的数据创建根结点  $r$ 。
- 2) 如果左子树存在且不空，将它的根结点链接为  $r$  的左孩子。
- 3) 如果右子树存在且不空，且与左子树不是同一棵树，将它的根结点链接为  $r$  的右孩子。但如果右子树和左子树是相同的，则将右子树的拷贝链接为  $r$  的右子树。
- 4) 如果左子树存在，且与调用 `initializeTree` 的对象树不是同一棵树，则将该子树的数据域 `root` 置为 `null`。
- 5) 如果右子树存在，且与调用 `initializeTree` 的对象树不是同一棵树，则将该子树的数据域 `root` 置为 `null`。

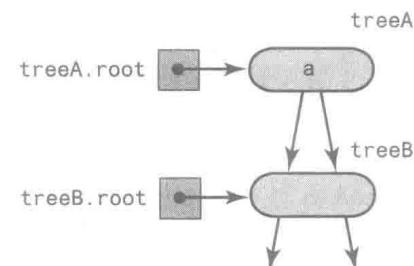


图 25-3 `treeA` 的子树是同一棵树

initializeTree 的实现如下：

```
private void initializeTree(T rootData, BinaryTree<T> leftTree,
 BinaryTree<T> rightTree)
{
 root = new BinaryNode<T>(rootData);
 if ((leftTree != null) && !leftTree.isEmpty())
 root.setLeftChild(leftTree.root);
 if ((rightTree != null) && !rightTree.isEmpty())
 {
 if (rightTree != leftTree)
 root.setRightChild(rightTree.root);
 else
 root.setRightChild(rightTree.root.copy());
 } // end if
 if ((leftTree != null) && (leftTree != this))
 leftTree.clear();
 if ((rightTree != null) && (rightTree != this))
 rightTree.clear();
} // end initializeTree
```



学习问题 2 initializeTree 实现的最后，可以置 rightTree 为 null，而不是调用 clear 吗？请解释。

## 访问方法和赋值方法

公有方法 setRootData、getRootData、isEmpty 和 clear 很容易实现。除了这些方法，我们还定义了几个保护方法——setRootNode 和 getRootNode——实现子类时它们很有用。这些方法的实现如下所示。EmptyTreeException 是我们定义的运行时异常类。25.8

```
public void setRootData(T rootData)
{
 root.setData(rootData);
} // end setRootData
public T getRootData()
{
 if (isEmpty())
 throw new EmptyTreeException();
 else
 return root.getData();
} // end getRootData
public boolean isEmpty()
{
 return root == null;
} // end isEmpty
public void clear()
{
 root = null;
} // end clear
protected void setRootNode(BinaryNode<T> rootNode)
{
 root = rootNode;
} // end setRootNode
protected BinaryNode<T> getRootNode()
{
 return root;
} // end getRootNode
```

## 计算高度和结点个数

25.9 BinaryTree 内的方法。方法 getHeight 和 getNumberOfNodes 比上段中给出的方法更加令人感兴趣。虽然这些必要的计算可以在类 BinaryTree 内执行，但在类 BinaryNode 内执行更简单些。所以 BinaryTree 类的下列方法调用了 BinaryNode 内的类似方法：

```
public int getHeight()
{
 int height = 0;
 if (root != null)
 height = root.getHeight();
 return height;
} // end getHeight

public int getNumberOfNodes()
{
 int numberOfNodes = 0;
 if (root != null)
 numberOfNodes = root.getNumberOfNodes();
 return numberOfNodes;
} // end getNumberOfNodes
```

现在完成了 BinaryNode 类中的 getHeight 和 getNumberOfNodes 方法。

25.10 BinaryNode 内的方法。在 BinaryNode 中，方法 getHeight 返回以调用该方法的结点为根的子树的高度。类似地，getNumberOfNodes 返回同一棵子树中的结点个数。

公有方法 getHeight 可以调用私有的递归方法 getHeight，后者带有一个结点作为形参。以一个结点为根的树的高度，等于 1——结点本身——再加上该结点最高子树的高度。故代码实现如下：

```
public int getHeight()
{
 return getHeight(this); // Call private getHeight
} // end getHeight

private int getHeight(BinaryNode<T> node)
{
 int height = 0;
 if (node != null)
 height = 1 + Math.max(getHeight(node.getLeftChild()),
 getHeight(node.getRightChild()));
 return height;
} // end getHeight
```

可以使用相同的机制实现 getNumberOfNodes，不过我们介绍另外一种方法。以给定结点为根的树中的结点个数，等于 1——结点本身——再加上左子树中的结点个数和右子树中的结点个数。故递归代码实现如下：

```
public int getNumberOfNodes()
{
 int leftNumber = 0;
 int rightNumber = 0;
 if (leftChild != null)
 leftNumber = leftChild.getNumberOfNodes();
 if (rightChild != null)
 rightNumber = rightChild.getNumberOfNodes();
 return 1 + leftNumber + rightNumber;
} // end getNumberOfNodes
```

## 遍历

递归地遍历二叉树。第 24 章描述了遍历二叉树中所有结点的 4 种次序：中序、前序、后序和层序。例如，中序遍历访问根左子树中的所有结点，然后访问根，最后访问根右子树中的所有结点。因为中序遍历访问子树中的结点时仍使用中序遍历，所以这个描述是递归的。

25.11

可以在类 `BinaryTree` 中添加一个递归方法来完成中序遍历。不过这样的一个方法必须要对它所访问的每个结点内的数据做些事情。为简单起见，我们只显示数据，尽管实现 ADT 的类一般不应执行输入和输出。

对于递归处理子树的方法，它需要子树的根作为参数。为了对客户隐藏细节，让递归方法是私有的，且从一个无参数的公有方法来调用它。故方法实现如下：

```
public void inorderTraverse()
{
 inorderTraverse(root);
} // end inorderTraverse

private void inorderTraverse(BinaryNode<T> node)
{
 if (node != null)
 {
 inorderTraverse(node.getLeftChild());
 System.out.println(node.getData());
 inorderTraverse(node.getRightChild());
 } // end if
} // end inorderTraverse
```

可以为前序遍历和后序遍历实现类似的方法。



**学习问题 3** 使用图 25-4 中的二叉树，跟踪方法 `inorderTraverse` 的执行。显示的数据是什么？

**学习问题 4** 实现以前序遍历次序显示二叉树中数据的递归方法 `preorderTraverse`。

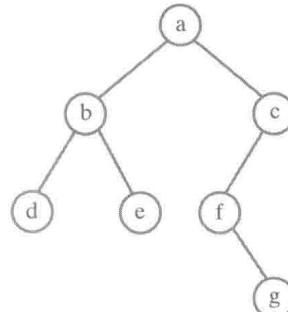


图 25-4 二叉树

**注：**一般地，实现 ADT 的类内的方法不应该执行输入和输出。此处我们这样处理是为了简化遍历方法。不过，像 `inorderTraverse` 这样的方法不用真的显示树中的数据，而是返回由数据组成的一个串。使用这样一棵树的客户就可以使用下面的语句显示这个串：

```
System.out.println(myTree.inorderTraverse());
```

**使用迭代器进行遍历。**像 `inorderTraverse` 这样的方法不难实现，但该方法在遍历时仅显示数据。另外，一旦调用方法就会完成一次全部的遍历。为给客户提供更大的灵活性，可以将遍历定义为迭代器。这种方式下，在访问过程中，客户可做的事情就不仅仅是显示数据这么简单了，还可以在每次访问时加以控制。

25.12

回忆一下，Java 的接口 `Iterator` 中声明了方法 `hasNext` 和 `next`。这些方法可让客户在遍历过程中的任何时刻获取当前结点中的数据。就是说，客户可以获取一个结点的数据，对它进行操作，还可能做点其他事情，然后再获取迭代中下一个结点中的数据。

如果看第 24 章段 24.20 中有关 `BinaryTreeInterface` 的内容，可以明白，如 `BinaryTree` 这样的实现了 `BinaryTreeInterface` 的任何类，也必须定义接口 `TreeIterator`。

Interface 中的方法。例如，在类 `BinaryTree` 内的方法 `getInorderIterator` 实现如下：

```
public Iterator<T> getInorderIterator()
{
 return new InorderIterator();
} // end getInorderIterator
```

与第 24 章中的处理一样，将类 `InorderIterator` 定义为 `BinaryTree` 的私有内部类。

迭代器在遍历过程中必须能暂停。这暗示着不能用递归来实现。第 9 章显示了如何使用栈来替代递归。这正是我们现在要做的。

25.13

**中序遍历的迭代版本。**定义一个迭代器之前，先考虑执行中序遍历的迭代方法。这个方法比构造迭代器要简单一点儿，但步骤差不多。

图 25-5 显示了图 25-4 中的树，及使用栈来执行中序遍历时的结果。从将根 `a` 入栈开始。然后一直向左遍历到尽头，将每个结点入栈。因为 `d` 没有左孩子，所以从栈中弹出它并显示它。因为 `d` 没有右孩子结点，所以再次出栈并显示 `b`。`b` 有右孩子 `e`，将 `e` 入栈。因为 `e` 没有孩子，将它出栈并显示它。继续这个过程，直到访问过所有结点时为止——就是说，直到栈为空且当前结点是 `null` 时为止。

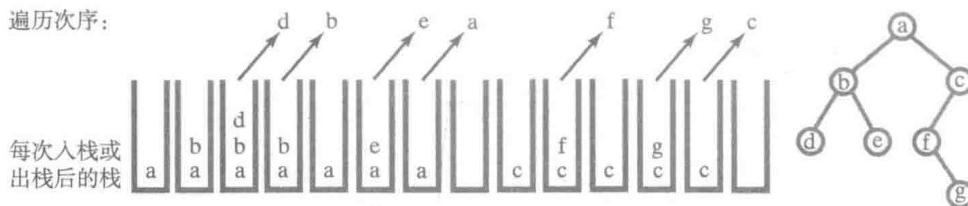


图 25-5 使用栈来执行二叉树的中序遍历

下面是执行中序遍历的迭代方法。

```
public void iterativeInorderTraverse()
{
 StackInterface<BinaryNode<T>> nodeStack = new LinkedStack<>();
 BinaryNode<T> currentNode = root;
 while (!nodeStack.isEmpty() || (currentNode != null))
 {
 // Find leftmost node with no left child
 while (currentNode != null)
 {
 nodeStack.push(currentNode);
 currentNode = currentNode.getLeftChild();
 } // end while
 // Visit leftmost node, then traverse its right subtree
 if (!nodeStack.isEmpty())
 {
 BinaryNode<T> nextNode = nodeStack.pop();
 // Assertion: nextNode != null, since nodeStack was not empty
 // before the pop
 System.out.println(nextNode.getData());
 currentNode = nextNode.getRightChild();
 } // end if
 } // end while
} // end iterativeInorderTraverse
```



学习问题 5 使用第 24 章图 24-14 中的二叉树跟踪前一个方法的执行。

25.14

私有类 InorderIterator。现在将中序遍历实现为一个迭代器。将前一个方法 iterativeInorderTraverse 中的逻辑分散到迭代器的构造方法及 hasNext 和 next 方法中。栈和变量 currentNode 是迭代器类中的数据域。方法 next 将 currentNode 前推，必要时一直入栈，最终出栈时返回结点中的数据，那就是迭代的下一项。私有内部类 InorderIterator 的实现列在程序清单 25-3 中。

### 程序清单 25-3 私有内部类 InorderIterator

```

1 private class InorderIterator implements Iterator<T>
2 {
3 private StackInterface<BinaryNode<T>> nodeStack;
4 private BinaryNode<T> currentNode;
5
6 public InorderIterator()
7 {
8 nodeStack = new LinkedStack<>();
9 currentNode = root;
10 } // end default constructor
11
12 public boolean hasNext()
13 {
14 return !nodeStack.isEmpty() || (currentNode != null);
15 } // end hasNext
16
17 public T next()
18 {
19 BinaryNode<T> nextNode = null;
20
21 // Find leftmost node with no left child
22 while (currentNode != null)
23 {
24 nodeStack.push(currentNode);
25 currentNode = currentNode.getLeftChild();
26 } // end while
27
28 // Get leftmost node, then move to its right subtree
29 if (!nodeStack.isEmpty())
30 {
31 nextNode = nodeStack.pop();
32 // Assertion: nextNode != null, since nodeStack was not empty
33 // before the pop
34 currentNode = nextNode.getRightChild();
35 }
36 else
37 throw new NoSuchElementException();
38
39 return nextNode.getData();
40 } // end next
41
42 public void remove()
43 {
44 throw new UnsupportedOperationException();
45 } // end remove
46 } // end InorderIterator

```

迭代的前序、后序和层序遍历。图 25-6 显示使用栈来完成图 25-4 所示树的前序遍历和后序遍历的结果。迭代的后序遍历——和前面的迭代中序遍历一样——将递归中的每次递归调用替换为一个 push 操作，将每次访问替换为一次 pop 操作。不过，迭代的前序遍历中，结点的孩子入栈的次序与递归前序遍历中递归调用的次序相反，这样才能以正确的顺序访问

25.15

结点。最后，迭代的层序遍历中用到队列而不是栈。遍历时先将根入队列。当队列不空时，遍历方法出队一个结点，访问该结点，并将该结点的孩子结点入队列。图 25-7 显示了使用队列对同一棵树执行层序遍历的结果。将迭代类的实现留作练习。

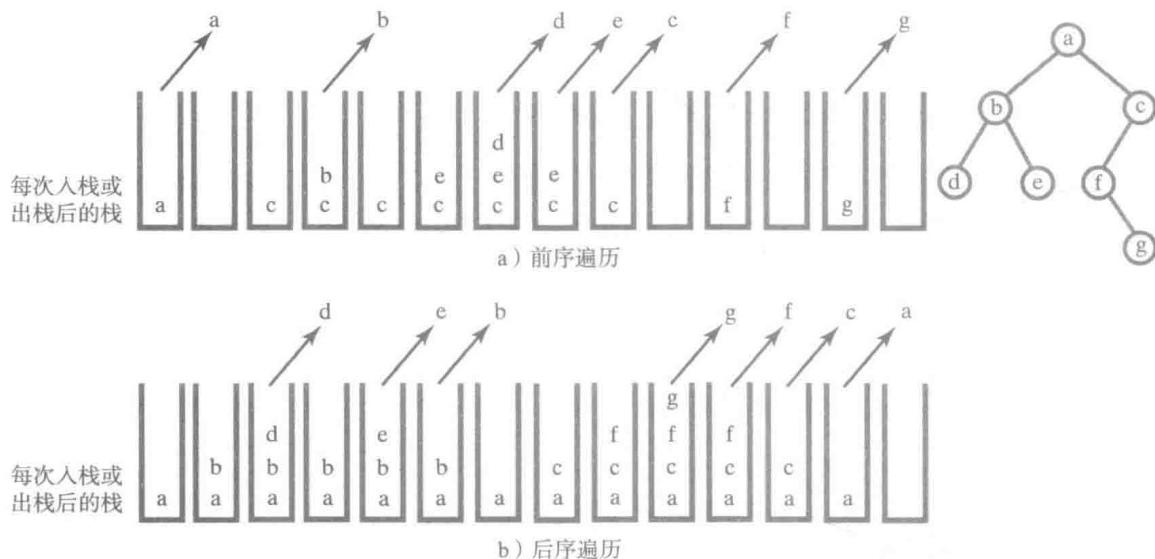


图 25-6 使用栈以前序和后序遍历二叉树

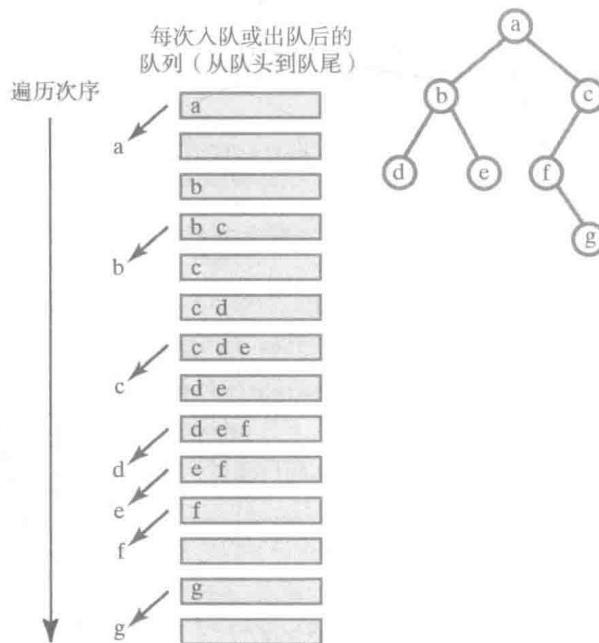


图 25-7 使用队列层序遍历二叉树



**程序设计技巧：**还没有遍历完整个二叉树的迭代器对象，可能会遇到树已改变这种不利情况。



**注：**如果访问一个结点是  $O(1)$  的，则对含  $n$  个结点的二叉树的完整遍历，递归和迭代的实现版本都是  $O(n)$  的。

## 表达式树的实现

在第 24 章中已经了解了表达式树是一棵表示代数表达式的二叉树。图 24-15 给出了这些树的几个示例。使用段 24.24 中给出的算法，可以计算这类树中表达式的值。25.16

我们可以从用于二叉树的接口进行派生，添加方法 evaluate 的声明，从而为表达式树定义一个接口，如程序清单 25-4 所示。因为可以将组成表达式的各成份看成是字符串，所以假定表达式树结点中的数据是串。

**程序清单 25-4** 用于表达式树的接口

```

1 package TreePackage;
2 public interface ExpressionTreeInterface
3 extends BinaryTreeInterface<String>
4 {
5 /** Computes the value of the expression in this tree.
6 * @return The value of the expression. */
7 public double evaluate();
8 } // end ExpressionTreeInterface

```

表达式树是一棵二叉树，所以可以从 BinaryTree 派生表达式树类。在派生类中定义方法 evaluate。类 ExpressionTree 的部分内容列在程序清单 25-5 中。25.17

**程序清单 25-5** 类 ExpressionTree

```

1 package TreePackage;
2 public class ExpressionTree extends BinaryTree<String>
3 implements ExpressionTreeInterface
4 {
5 public ExpressionTree()
6 {
7 } // end default constructor
8
9 public double evaluate()
10 {
11 return evaluate(getRootNode());
12 } // end evaluate
13
14 private double evaluate(BinaryNode<String> rootNode)
15 {
16 double result;
17 if (rootNode == null)
18 result = 0;
19 else if (rootNode.isLeaf())
20 {
21 String variable = rootNode.getData();
22 result = getValueOf(variable);
23 }
24 else
25 {
26 double firstOperand = evaluate(rootNode.getLeftChild());
27 double secondOperand = evaluate(rootNode.getRightChild());
28 String operator = rootNode.getData();
29 result = compute(operator, firstOperand, secondOperand);
30 } // end if
31
32 return result;
33 } // end evaluate
34
35 private double getValueOf(String variable)
36 {

```

```

37 } ...
38 } // end getValueOf
39
40 private double compute(String operator, double firstOperand,
41 double secondOperand)
42 {
43 ...
44 } // end compute
45 } // end ExpressionTree

```

公有方法 `evaluate` 调用递归的私有方法 `evaluate`。该私有方法又调用了私有方法 `getValueOf` 和 `compute`，以及 `BinaryNode` 类中定义的方法。方法 `getValueOf` 返回表达式中指定变量的数值，而 `compute` 返回给定的运算符运用于给定的两个操作数的计算结果。

注意到类 `BinaryNode` 中的方法对 `evaluate` 的实现是多么重要。为此，不想让类 `BinaryNode` 隐藏在类 `ExpressionTree` 中，而应该处于同一个包中。



**学习问题 6** 对第 24 章图 24-15c 所示的表达式树，跟踪方法 `evaluate` 的执行。假定  $a$  是 3， $b$  是 4 而  $c$  是 5。返回值是什么？

## 一般树

我们再来考虑一般树中结点的表示方法，以结束对树实现的讨论。不是要用这个结点来实现一般树，而是看看如何使用二叉树来表示一般树。

### 用于一般树的结点

25.18

因为二叉树中的结点仅能有两个孩子，所以每个结点含有两个指向孩子的引用就很合理。另外，为了测试设置和获取结点的每个孩子而需要的结点编号操作也是合理的。但当每个结点中有更多孩子时，使用这种处理方法并不便利。

可以为一般树定义一个结点，用引用指向一个包含孩子的对象，例如线性表或是向量，以满足孩子个数任意性的需要。例如，图 25-8 中的结点含有两个引用。一个引用指向数据对象，另一个引用指向一个孩子结点线性表。可以用线性表迭代器来访问这些孩子结点。

在程序清单 25-6 给出的用于一般树结点的接口中，`getChildrenIterator` 返回结点孩子的迭代器。另一个操作是将一个孩子添加到结点中，假定孩子间没有特定的次序。如果孩子间的次序很重要，则迭代器可以提供将新孩子插入在迭代的当前位置的操作。

#### 程序清单 25-6 用于一般树结点的接口

```

1 package TreePackage;
2 import java.util.Iterator;
3 interface GeneralNodeInterface<T>
4 {
5 public T getData();
6 public void setData(T newData);
7 public boolean isLeaf();
8 public Iterator<GeneralNodeInterface<T>> getChildrenIterator();
9 public void addChild(GeneralNodeInterface<T> newChild);
10 } // end GeneralNodeInterface

```

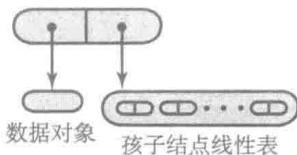


图 25-8 用于一般树的结点



注：对一般树操作的算法比对二叉树的要复杂得多，因为一般树中每个结点的孩子结点的个数是任意的。因为这个原因，一般树有时用二叉树来表示。注意，很多文件系统在设计时使用一般树来加快查找磁盘中目录和文件的速度。下一节讨论如何将一般树转为等价的二叉树。

## 使用二叉树表示一般树

抛开上面刚提出的实现方案，可以使用二叉树来表示一般树。例如，将图 25-9a 所示的一般树表示为一棵二叉树。作为中间步骤，我们用新边来连接结点，过程如下。让根 A 原来的孩子之一——本例中是 B——作为左孩子。然后从 B 到其兄弟 C 间及从 C 到另一个兄弟 D 间各画一条边，如图 25-9b 所示。类似地，对一般树中的每个父结点，让原孩子中的一个作为二叉树中的左孩子，同时在这些孩子结点之间用边相连。25.19

如果将图 25-9b 中的每个位于其兄弟结点右侧的结点，看作那个兄弟结点的右孩子，则得到一棵非传统形式的二叉树。画树时可以在保持连线的前提下，移动结点的位置，则得到熟悉的二叉树的样子，如图 25-9c 所示。25.20

遍历。来看看图 25-9a 所示一般树的几种不同的遍历结果，并与图 25-9c 所画的等价二叉树的遍历结果进行比较。一般树的遍历结果如下：

前序：A, B, E, F, C, G, H, I, D, J

后序：E, F, B, G, H, I, C, J, D, A

层序：A, B, C, D, E, F, G, H, I, J

二叉树的遍历结果如下：

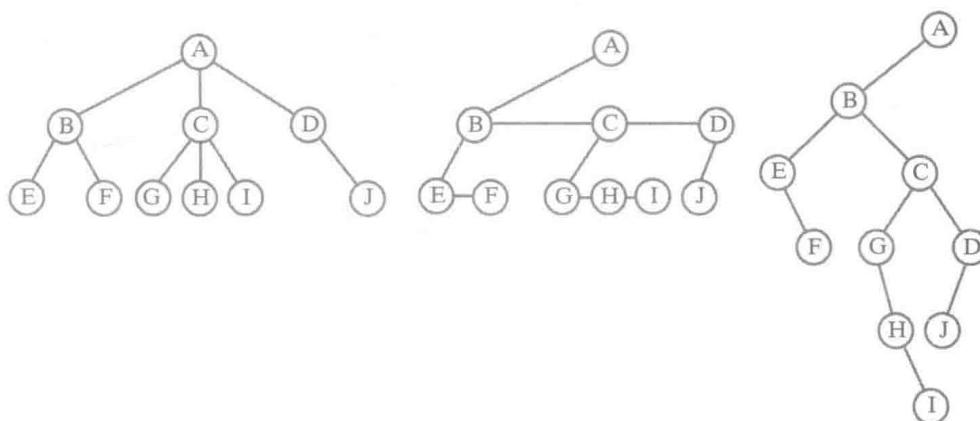
前序：A, B, E, F, C, G, H, I, D, J

后序：F, E, I, H, G, J, D, C, B, A

层序：A, B, E, C, F, G, D, H, J, I

中序：E, F, B, G, H, I, C, J, D, A

两棵树的前序遍历结果是一样的。一般树的后序遍历结果与二叉树的中序遍历结果是一样的。我们必须发现一种新的二叉树的遍历机制，以得到与一般树的层序遍历相同的结果。将这个工作留作练习 11。



a ) 一般树

b ) 等价的二叉树

c ) 同一棵二叉树的常规形式

图 25-9 一般树及其等价的二叉树的两个视图

学习问题 7 第 24 章图 24-1 所示的一般树能表示成一棵什么样的二叉树?



## 本章小结

- 二叉树中的结点是一个对象，其中的引用指向数据对象和树中的两个孩子结点。
- 二叉树的基本类中含有所有树公用的方法：`getRootData`、`getHeight`、`getNumberOfNodes`、`isEmpty`、`clear` 和几种遍历方法。基本类中还有一个方法，可以将已存在的二叉树的根和子树设置为给定的值。
- 如果结点类中有类似方法，则 `getHeight` 和 `getNumberOfNodes` 的实现更简单。
- 前序、后序和中序遍历的递归实现很简单。但要将遍历实现为迭代器则必须使用迭代方法，因为迭代器必须能在遍历过程中暂停。前序、后序和中序遍历中使用栈；层序遍历中使用队列。
- 可以从基本二叉树类派生特殊的二叉树类，例如表达式树类。
- 一般树中的结点是一个对象，它指向其孩子和一个数据对象。为了能容纳任意多个孩子，比如，结点可以指向一个线性表或是向量。迭代器可以提供对孩子的访问。这种实现方式下，结点中仅含有两个引用。
- 并不创建用于一般树的一般结点，而是使用二叉树来表示一般树。

## 程序设计技巧

- 当将 `BinaryTree` 的实例传给形参类型为 `BinaryTreeInterface` 的方法时，不需要进行转型。但是反过来是需要转型的。
- 还没有遍历完整个二叉树的迭代器对象，可能会遇到树已改变这种不利情况。

## 练习

1. 使用段 25.10 中用于实现 `getNumberOfNodes` 方法的机制，实现类 `BinaryNode` 中的 `getHeight` 方法。即 `getHeight` 不应该调用一个私有方法。
2. 使用段 25.10 中用于实现 `getHeight` 方法的机制，实现类 `BinaryNode` 中的 `getNumberOfNodes` 方法。即 `getNumberOfNodes` 应该调用一个私有方法。
3. 在段 25.11 中，学习问题 4 要求实现一个递归的前序遍历二叉树的方法。实现以后序遍历次序显示二叉树中数据的递归方法 `postorderTraverse`。
4. 使用图 25-10 中的二叉树，跟踪段 25.13 中给出的 `iterativeInorderTraverse` 迭代方法的执行。每次 `push` 和 `pop` 操作后显示栈的内容。
5. 前序遍历图 25-10 所示的二叉树，每次 `push` 和 `pop` 后显示栈的内容。再用后序遍历重做本题。
6. 层序遍历图 25-10 所示的二叉树，每次 `enqueue` 和 `dequeue` 后显示队列的内容。
7. 假定想为 `BinaryTree` 类创建一个方法，用来统计一个对象在树中出现的次数。方法头可能如下：

```
public int count(T anObject)
```

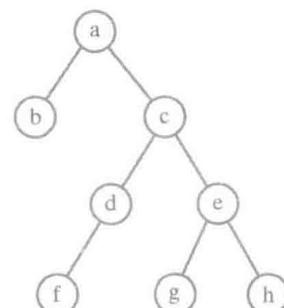


图 25-10 用于练习 4、练习 5、练习 6 和练习 17 的二叉树

- a. 使用同名的私有递归方法来写这个方法。  
 b. 使用二叉树的一个迭代器来写这个方法。  
 c. 比较前面两种实现版本的效率。
8. 使用第 24 章图 24-15d 所示的表达式树，跟踪段 25.17 所给的 `evaluate` 的执行。返回的值是什么？假定  $a$  是 2,  $b$  是 4,  $c$  是 5,  $d$  是 6 且  $e$  是 4。
9. 递归定义含  $n$  个结点的不同形状二叉树的可能个数。忽略结点中的内容。
10. 第 24 章中下列各图所示的一般树对应的二叉树分别是什么？  
 a. 图 24-5  
 b. 图 24-22
11. 给定一般树，考虑等价的二叉树。定义等价于一般树层序遍历的二叉树的遍历方法。
12. 已知二叉树的前序遍历和中序遍历能保证唯一定义这棵树。后序遍历和中序遍历也有相应的结论。  
 a. 画出有下列前序遍历和中序遍历的唯一的二叉树：  
 前序: A, B, D, E, C, F, G, H  
 中序: E, D, B, A, G, F, H, C  
 b. 画出有下列后序遍历和中序遍历的唯一二叉树：  
 后序: B, D, F, G, E, C, A  
 中序: B, A, D, C, F, E, G
13. 虽然由二叉树的前序遍历和中序遍历或由后序遍历和中序遍历，能唯一构造这棵树，但多棵二叉树可能有相同的前序遍历或是相同的后序遍历。  
 a. 给出有相同前序遍历的两棵不同二叉树的示例。  
 b. 给出有相同后序遍历的两棵不同二叉树的示例。  
 c. 给出其前序遍历与其后序遍历相同的一棵二叉树示例。
14. 想为类 `BinaryTree` 增加一个方法，它接受一个 `BinaryTreeInterface<T>` 对象实参，如果实参树与本二叉树有相同的结构则方法返回真。如果两棵树的结点有对应的位置，则两棵树的结果相同。方法头可能如下：
- ```
public boolean isStructurallyIdentical(BinaryTreeInterface<T> otherTree)
```
- 使用同名的私有递归方法写出该方法。
15. 重做练习 14，但写方法 `isIdentical`，如果实参树与该二叉树相同，则方法返回真。两棵树中的结点有对应的位置及相等的值，则两棵树相同。
16. 考虑练习 14 和练习 15。这两个方法 `isStructurallyIdentical` 和 `isIdentical`，哪个实现起来更困难一些？请解释。
17. 考虑有相同结构的两棵二叉树。一棵树中的结点的数据，可以不同于另一棵树中对应位置结点中的数据。编写代码，使用字典将第一棵树中的对象映射到第二棵树对应的对象。
18. 有时需要从树中的一个结点移到其父结点上。为此，必须为二叉树结点提供一个指向其父结点的引用。然后可以遍历从叶子到根的路径。重新设计二叉树的结点，让每个结点在指向其左孩子和右孩子的引用之外，还有指向其父结点的引用。哪些方法需要修改？
19. 二叉树的另一个表示方法是使用数组。树中的项按层序的方式放到数组中。例如，图 25-11 是表示图 25-10 所示二叉树的数组。注意，数组中的空白对应于树中缺失的结点。要表示高度为 4 的任意二叉树，数组要足够大。
 a. 下标 i 处保存的结点的孩子结点的下标是多少？
 b. 下标 i 处保存的结点的父结点的下标是多少？
 c. 这种表达方式的优缺点各是什么？

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| a | b | c | | | d | e | | | | | f | | g | h |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

图 25-11 练习 19 中表示图 25-10 所示二叉树的数组

20. 为类 `BinaryTree` 实现方法 `toString`。方法返回一个字符串，显示时将在平面中显示树形。忽略每个结点中的数据。例如，一棵树可能是下面这个样子的：



项目

- 使用图 25-6 和图 25-7 中的示例提出算法，实现用于二叉树前序、后序和层序遍历的迭代器类。先写迭代版本的遍历方法，类似于段 25.13 中所给的方法 `iterativeInorderTraverse`。
- 写一个区分 10 种不同动物的 Java 程序。程序应该玩一个类似于第 24 章段 24.26 中描述的猜猜看游戏。使用者从 10 个动物中选择一个，程序问一系列的问题，直到它能猜中动物时为止。
程序应该从使用者那里获取知识。如果程序猜错了，则要求使用者输入一个能区分正确动物和程序给出的错误答案的新问题。决策树应该用这个新问题来更新。
为本项目写两个方案。第一个应该使用 `DecisionTree` 的实例。第二个方案应该使用派生于 `DecisionTree` 的 `GuessingGameTree` 类。
- 完成段 25.17 开始的表达式树的实现。添加两个构造方法，由给定的后缀表达式或中缀表达式创建表达式树。第 24 章的项目 4 和项目 5 要求设计算法来完成这两个任务。为了简化 `getValueOf` 方法，可以限定变量的选择并给它们具体的值。
- 考虑练习 18 描述的为二叉树重新设计的结点。给结点添加一个额外的数据域，来记录以那个结点为根的子树的高度。修改二叉树中实现的所有方法，树的结构一旦改变，立即更新高度域。
- 第 24 章的练习 22 要求为一般树开发接口 `GeneralTreeInterface`。编写实现 `GeneralTreeInterface` 接口的类 `GeneralTree`。
 - 使用二叉树表示一般树，如段 25.19 所描述的。为前序遍历和后序遍历实现迭代器。作为额外的挑战，实现层序遍历迭代器。
 - 重做问题 a，但不使用二叉树。而是定义类 `GeneralNode`，它实现了程序清单 25-6 中给出的接口 `GeneralNodeInterface`。使用 `GeneralNode` 对象来定义一般树。
- 有些二叉树的实现不使用 `null` 表示没有孩子的情况。相反，它们使用指向唯一哑结点的引用。指向空树的引用也是指向该同一哑结点。按这种方式修改 `BinaryTree` 的实现。
- 实现练习 19 描述的使用数组表示树的类 `ArrayBinaryTree`。
- 实现第 24 章项目 7 设计的拼写检查器。
- 为术语表创建字典，如第 21 章项目 13 所描述的。不是使用基于数组的有序线性表，而是使用 26 叉树来表示术语。第 24 章项目 7 中的图 24-27 图示了这样的一棵树，不过那棵树是用来进行拼写检查的。为了让那棵树满足本项目的需求，填充结点中放置术语的定义，而不是仅用作术语拼写的标志。
- 哈夫曼编码（Huffman coding）是压缩数据长度的一项技术。例如，文本文档的 `.zip` 文件是原文档的压缩版。这种所谓的无损数据压缩就是哈夫曼编码的结果。

尽管 ASCII 和 Unicode 使用定长的位串——分别为 8 位和 16 位——表示符号，但哈夫曼编码

是变长的。这些编码以给定数据集中字符的出现频度为基础。出现频繁的符号比出现不太频繁的符号有更短的编码。二叉树——称为哈夫曼树 (huffman tree) ——用来产生这些编码。

例如，对仅由字符 A 到 E 组成的某些文本进行编码。假定这些字符出现的频度如下：A：12 次；B：3 次；C：1 次；D：9 次，E：15 次。我们需要根据这些字符的出现频度降序重排这些字符。为此，将每个字符与它的出现频度相关联，将这些数据对放入集合，例如一个有序表或是优先队列。结果如图 25-12a 所示。现在删除具有最低频度的两个项，让它们作为二叉树的叶子。这些叶子的父结点是含有叶结点频度之和的一个结点，如图 25-12b 所示。因为父结点中仅含有频度，故结点的字符部分是 null，图中显示为 ·。现在将父结点的内容添加到线性表或队列中，显示在图 25-12b 中树的右侧。

当从线性表中删除接下来的两项时，创建了包含 D 9 的新叶结点，并将其添加到已有的树中，新树根中含有两个孩子的频度之和。结果如图 25-12c 所示。注意，新树根结点要放到剩余数据中的正确位置。图 25-12d 和图 25-12e 说明了这个过程余下的步骤。

图 25-12f 显示得到的二叉树，其左孩子链和右孩子链分别标记为 0 和 1。为了给字符编码，从叶开始在树中遍历到根结点。根据你走的是左分支还是右分支，按逆序记录下 0 和 1。本例中哈夫曼编码如下：A 是 10，B 是 1101，C 是 1100，D 是 111 而 E 是 0。要解码哈夫曼编码，从根结点到叶结点遍历树，对遇到的每个 0 走左分支，遇到的每个 1 走右分支。

写一个程序读入字母数据的文本文件，创建一棵哈夫曼树，用树来压缩这个文件。然后程序还应该能使用你的树，对压缩文件进行解码。

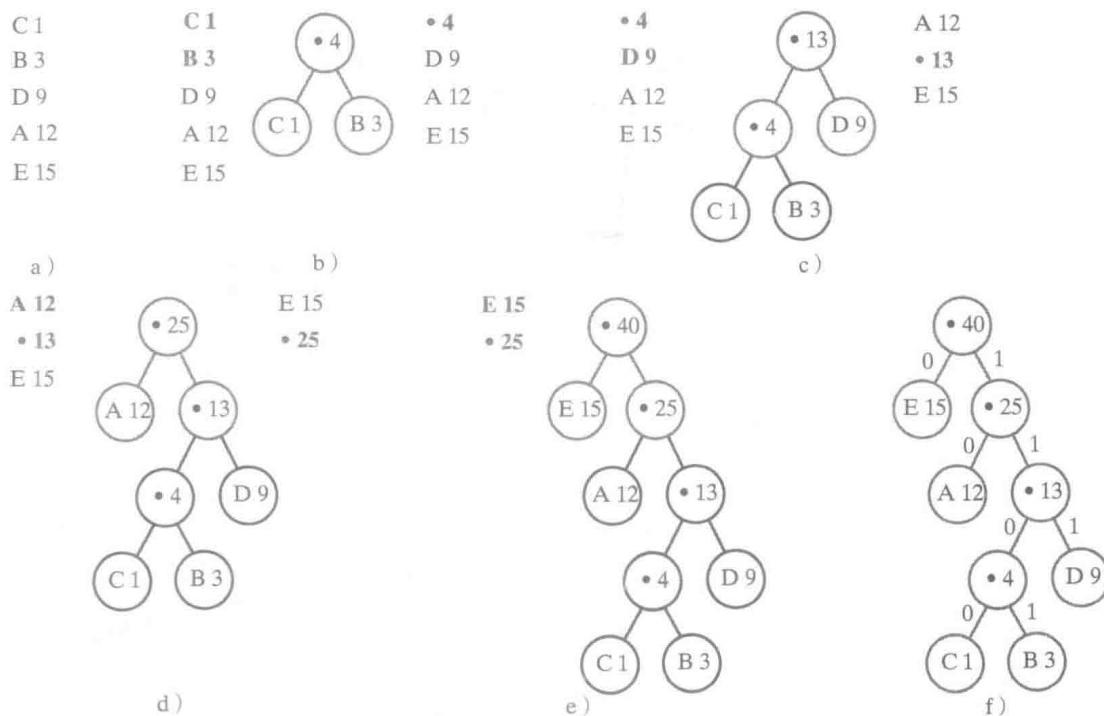


图 25-12 用于哈夫曼编码的二叉树的创建步骤

11. 紧接在本章之后的 Java 插曲 9，介绍对象的克隆。克隆是另外一个副本或拷贝。本章定义的类 `BinaryNode` 中，定义了方法 `copy` 来复制结点。(见段 25.5。) 这个方法用在段 25.7 的 `BinaryTree` 类的 `initializeTree` 方法中。Java 插曲 9 描述了如何用方法 `clone` 来替代方法 `copy`。`clone` 方法复制结点。修改 `BinaryTree` 类中的 `initializeTree` 方法，让它调用 `clone` 替代 `copy`。然后将 `clone` 方法添加到 `BinaryTree` 类中。

克 隆

先修章节：Java 插曲 5、Java 插曲 6、第 17 章、第 25 章、附录 B、附录 C

Java 插曲 6 的段 J6.4 中创建了有序的可变对象线性表。不幸的是，这个线性表的客户能够修改对象，使得它们不再有序。正如你所见的，一种办法是在有序表中总是放置不可变对象。一个更复杂的方法是，有一个拷贝了客户对象的线性表。然后让这个线性表来控制客户能或不能进行拷贝。本插曲研究如何对一个对象进行拷贝或克隆（clone）。

可克隆的对象

J9.1 在 Java 中，克隆是对象的拷贝。通常，我们仅克隆可变对象。因为共享一个不可变对象是安全的，克隆常常没什么必要。

类 `Object` 含有一个保护方法 `clone`，它返回对象的拷贝。方法有下列方法头：

```
protected Object clone() throws CloneNotSupportedException
```

因为 `clone` 方法是保护的，且因为 `Object` 是所有其他类的超类，所以任意方法的实现中都可以含有调用

```
super.clone()
```

但类的客户不能调用 `clone`，除非类重写了它且将它声明为公有的。对对象进行拷贝可能是昂贵的，所以你可能不想让类去做这件事情。让 `clone` 成为一个保护方法，Java 的设计者迫使你对克隆要再思量一下。

 **程序设计技巧：**并不是所有的类都有一个公有的 `clone` 方法。实际上，大多数类，包括只读的类都没有这个方法。

J9.2 如果想让你的类含有一个公有的 `clone` 方法，类就必须实现 Java 接口 `Cloneable` 来声明这件事，这个接口在 Java 类库的 `java.lang` 包中。这种类有如下的头部：

```
public class MyClass implements Cloneable
{ . . . }
```

接口 `Cloneable` 很简单

```
public interface Cloneable
{
} // end Cloneable
```

如你所见，接口是空的。它没有声明方法，专门用来表示一个类实现了 `clone`。如果在类定义中忘记写 `implements Cloneable` 了，然后使用类的实例来调用 `clone`，则会出现 `CloneNotSupportedException` 异常。这个结果起初可能会让你疑惑，特别是如果你确实实现了 `clone` 的情况下。

 **程序设计技巧：**如果程序产生了异常 `CloneNotSupportedException`，虽然在你的类中实现了方法 `clone`，不过很可能是在类定义中忘记写 `implements Cloneable` 了。

注: Cloneable 接口

空的 `Cloneable` 接口不是一个典型的接口。实现它的类表示，它将提供一个公有的 `clone` 方法。因为 Java 设计者想提供 `clone` 方法的默认实现，故他们将它放在 `Object` 类中，而没有放在接口 `Cloneable` 中。但因为设计者不想让每个类都自动拥有一个公有的 `clone` 方法，所以他们让 `clone` 是一个保护方法。

注: 克隆

克隆不应是每个类都能进行的操作。如果你想让你的类具有这个功能，则必须：

- 声明你的类实现了 `Cloneable` 接口。
- 在你的类中将从 `Object` 类继承的保护方法 `clone`，重写为公有版本。

示例：克隆一个 Name 对象。让我们为附录 B 的段 B.16 中的 `Name` 类添加 `clone` 方法。J9.3

```
public class Name implements Cloneable
```

类 `Name` 内的公有方法 `clone`，必须通过执行 `super.clone()` 来调用其超类的 `clone` 方法。因为 `Name` 的超类是 `Object`，所以 `super.clone()` 调用 `Object` 的保护方法 `clone`。`Object` 的 `clone` 方法可能会抛出一个异常，所以我们必须将每个调用都包含在一个 `try` 块中，并写 `catch` 块来处理异常。方法最后的动作应该是返回被克隆的对象。

所以 `Name` 的 `clone` 方法应该是下面这个样子的。

```
public Object clone()
{
    Name theCopy = null;
    try
    {
        theCopy = (Name)super.clone(); // Object can throw an exception
    }
    catch (CloneNotSupportedException e)
    {
        System.err.println("Name cannot clone: " + e.toString());
    }
    return theCopy;
} // end clone
```

因为 `super.clone()` 返回 `Object` 的一个实例，故我们将这个实例转型为 `Name`。毕竟，我们正在创建一个 `Name` 对象作为克隆。`return` 语句按需将 `theCopy` 隐式转型为 `Object`。但为什么不将 `theCopy` 的数据类型声明为 `Object` 从而避免转型呢？这样做，我们就不能在 `clone` 方法内使用 `theCopy` 来调用 `Name` 的方法了。因为在这里，`theCopy` 并没有做这件事情，所以它的数据类型可以是 `Object` 的。但并不赞成将 `theCopy` 声明为 `Object` 的。我们写的这个 `clone` 方法，具有更常用的方式。

`Object` 的 `clone` 方法能抛出的异常是 `CloneNotSupportedException`。因为已经为我们的 `Name` 类写了 `clone` 方法，所以这个异常永远不会发生。即使这样，当调用 `Object` 的 `clone` 方法时，仍必须使用 `try` 和 `catch` 块。在 `catch` 块中不是写 `println` 语句，而是写更简单的语句

```
throw new Error("Name cannot clone: " + e.toString());
```

! 程序设计技巧：每个公有的 `clone` 方法必须通过执行 `super.clone()`——一般地作为第一个动作——来调用基类的 `clone` 方法。这个调用必须放在 `try` 块中，哪怕永远不会出现 `CloneNotSupportedException`。最终，调用的是 `Object` 的保护的 `clone` 方法。

! 程序设计技巧：当 `Name` 的 `clone` 方法调用 `Object` 的保护 `clone` 方法时，会返回一个 `Object` 的实例。通过将这个实例转型为 `Name`，你可以肯定，即使 `Name` 的 `clone` 方法返回的是 `Object` 的实例，客户也能将返回值转型回 `Name` 对象。要知道，如果返回值不能转型为 `Name` 对象，则 Java 将抛出 `ClassCastException`。

! 程序设计技巧：正像你应该用名字称呼一个人，而不是喊“嗨对象！”一样，应该将 Java 对象声明为其实际的数据类型或泛型，而不是 `Object` 类型。

J9.4

两种克隆方法。当方法 `clone` 克隆一个对象时，它拷贝对象的数据域。当数据域本身又是一个对象时，可以用下列两种办法之一进行拷贝：

- 可以拷贝指向数据对象的引用，并与克隆共享对象，如图 JI9-1a 所示。这个克隆是浅克隆（shallow clone）。
- 可以拷贝对象本身，如图 JI9-1b 所示。当一个被克隆的对象又有原始组件对象的克隆时，克隆是深克隆（deep clone）。

! 注：`Object` 的 `clone` 方法返回一个浅克隆。

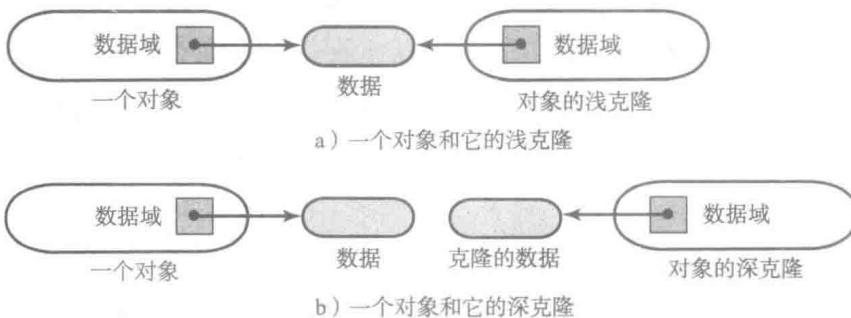


图 JI9-1 对象的两种克隆，浅克隆和深克隆

J9.5

Name 的克隆是浅的。类 `Name` 有数据域 `first` 和 `last`，它们都是 `String` 的实例。每个域含有一个指向字符串的引用。当 `clone` 方法执行 `super.clone()` 时，拷贝的正是这些引用。例如，图 JI9-2 说明了下列语句创建的对象：

```
Name april = new Name("April", "Jones");
Name twin = (Name)april.clone();
```

克隆 `twin` 是浅克隆，因为没有拷贝姓氏与名字中的字符串。

浅克隆对类 `Name` 已经足够好了。回忆一下，`String` 的实例是不可变的。让 `Name` 的实例和其克隆共享相同的字符串不会有问题，因为没有人能改变字符串。这是个好消息，因为与 Java 提供的许多类一样，`String` 没有 `clone` 方法。所以，如果我们要改变克隆的姓氏，需要写语句

```
twin.setLast("Smith");
```

`twin` 的姓氏将是 Smith，但 `april` 的仍是 Jones，如图 JI9-3 所示。即 `setLast` 改变的是 `twin` 的数据域 `last`，故它指向另一个字符串 Smith。它不改变 `april` 的数据域 `last`，所以它仍指向 Jones。

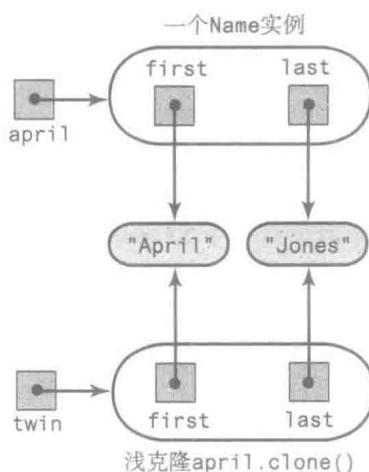


图 JI9-2 Name 的实例和其浅克隆

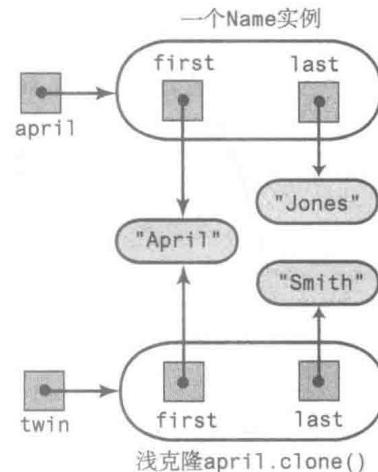


图 JI9-3 语句 `twin.setLast("Smith")`
改变了一个数据域之后的克隆 `twin`



程序设计技巧：指向不可变对象的数据域的浅拷贝，对于克隆来说通常就够用了。共享一个不可变对象常常是安全的。



示例：创建单一数据域的深克隆。有时浅克隆是不合适的。如果一个类有可变对象作为数据域，则你必须克隆对象，且不能简单拷贝它们的引用。例如，现在为附录 C 的段 C.2 中遇到的类 `Student` 添加一个 `clone` 方法。添加必需的 `implements` 子句后，类有下列形式：

```
public class Student implements Cloneable
{
    private Name   fullName;
    private String id;
    <此处是构造方法和方法setStudent, setName, setId, getName, getId及
     toString>
    .
} // end Student
```

J9.6

因为类 `Name` 有设置方法，故数据域 `fullName` 是可变对象。所以，肯定要在 `Student` 的 `clone` 方法的定义中克隆 `fullName`。能这样做是因为我们在段 J9.3 中为 `Name` 类添加了一个 `clone` 方法。因为 `String` 是只读类，`id` 是不可变的，故没有必要克隆它。所以，我们能为 `Student` 类定义一个 `clone` 方法，如下：

```
public Object clone()
{
    Student theCopy = null;
    try
    {
        theCopy = (Student)super.clone(); // Object can throw an exception
    }
```

```

    catch (CloneNotSupportedException e)
    {
        throw new Error(e.toString());
    }
    theCopy.fullName = (Name)fullName.clone();
    return theCopy;
} // end clone
}

```

在执行 `super.clone()` 后，调用 `Name` 的公有方法 `clone`，来克隆可变数据域 `fullName`。后一个调用不需要写在 `try` 块内。只有 `Object` 的 `clone` 方法含有一条 `throws` 子句。

图 JI9-4 说明了 `Student` 的一个实例及这个方法返回的克隆。如你所见，指向学生全名的 `Name` 对象被拷贝了，但表示姓氏及名字的字符串，及 ID 号都没有被拷贝。

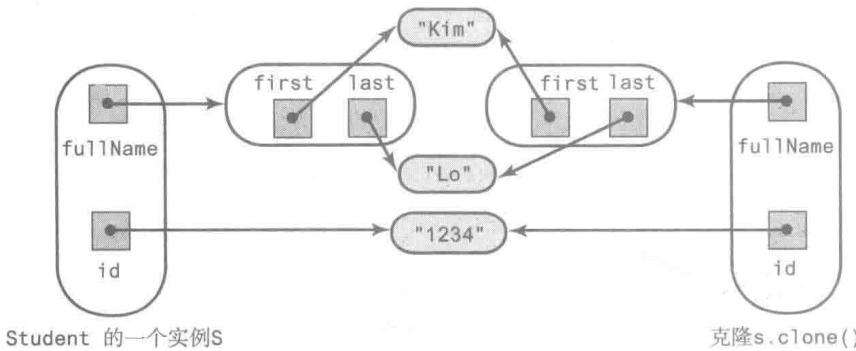


图 JI9-4 `Student` 的一个实例和它的克隆，包括 `fullName` 的一个深拷贝

如果没有克隆数据域 `fullName`——即没写语句

```
theCopy.fullName = (Name)fullName.clone();
```

则学生的全名会被原实例及其克隆一起共享。图 JI9-5 说明了这种情况。

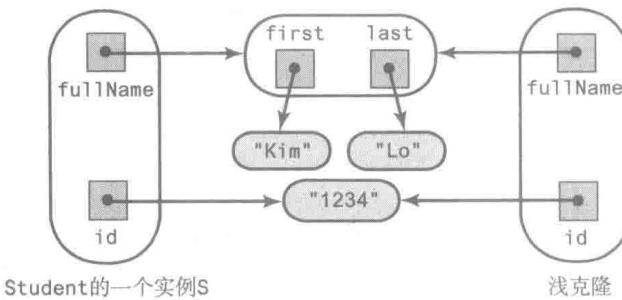


图 JI9-5 `Student` 的一个实例和它的克隆，包括 `fullName` 的一个浅拷贝



学习问题 1 假定 `x` 是 `Student` 的一个实例，而 `y` 是它的克隆，即

```
Student y = (Student)x.clone();
```

a. 如果执行下列语句改变 `x` 的姓氏：

```
Name xName = x.getName();
xName.setLast("Smith");
```

那么 `y` 的姓氏会改变吗？请解释之。

b. 如果在 `Student` 的 `clone` 方法内没有克隆 `fullName`，则修改 `x` 的姓氏也会改变 `y` 的姓氏吗？请解释之。



注：在每个公有 `clone` 方法内，一般地执行下列任务：

- 写 `super.clone()` 来调用超类的 `clone` 方法。
- 将对 `clone` 的这个调用放在 `try` 块中，且写一个 `catch` 块来处理可能出现的异常 `CloneNotSupportedException`。如果 `super.clone()` 调用一个公有 `clone` 方法，则可以跳过这一步。
- 如果可能，克隆 `super.clone()` 返回的对象的可变数据域。
- 返回克隆。



示例：克隆 `CollegeStudent` 对象。现在为 `Student` 的一个子类添加 `clone` 方法。J9.7

附录 C 的段 C.8 中定义了这样的一个子类，名叫 `CollegeStudent`。其定义中的 `implements` 子句是可选的，因为它是从一个可克隆类派生的：

```
public class CollegeStudent extends Student implements Cloneable
{
    private int year; // Year of graduation
    private String degree; // Degree sought
    <此处是构造方法和方法setStudent, setYear, getYear, setDegree, getDegree,
     toString和clone>
    . . .
} // end CollegeStudent
```

`CollegeStudent` 对象的数据域是基本类型的值和不可变对象，故它们不需要被克隆。所以为 `CollegeStudent` 添加的 `clone` 的定义如下：

```
public Object clone()
{
    CollegeStudent theCopy = (CollegeStudent)super.clone();
    return theCopy;
} // end clone
```

方法必须调用 `Student` 的 `clone` 方法，通过执行 `super.clone()` 来完成的。注意，因为 `Student` 的 `clone` 方法不抛出异常，所以调用它时不需要 `try` 块。如果 `CollegeStudent` 定义了需要克隆的域，则应该在 `return` 语句之前克隆它们。

克隆一个数组

第 11 章看到的类 `AList` 使用数组来实现 ADT 线性表。假定我们想为这个类添加 `clone` 方法。J9.8

当对线性表进行拷贝时，`clone` 方法必须拷贝数组及在其中的所有对象。所以，线性表中的对象也要有 `clone` 方法。回忆一下，`AList` 为其保存的对象定义了泛型 `T`。若 `AList` 的开头是这样的

```
public class AList<T extends Cloneable> . . . // Incorrect
```

将不能正确工作，因为接口 `Cloneable` 是空的。

相反，我们定义新的接口，声明一个公有方法 `clone` 来重写 `Object` 中的保护方法。

```
public interface Copyable extends Cloneable
{
    public Object clone();
} // end Copyable
```

这样，`AList` 的开头可以是下列语句之一：

- public class AList<T extends Copyable> implements ListInterface<T>, Cloneable
- public class AList<T extends Copyable> implements ListInterface<T>, Copyable
- public class AList<T extends Copyable> implements CloneableListInterface<T>

其中 `CloneableListInterface` 的定义如下所示。

```
public interface CloneableListInterface<T>
    extends ListInterface<T>, Copyable // Or Cloneable
{
} // end CloneableListInterface
```

符号

```
AList<T extends Copyable>
```

要求线性表中的对象属于实现了接口 `Copyable` 的类。

注意，`CloneableListInterface` 派生于两个接口 `ListInterface` 及 `Copyable` 或 `Cloneable`。如序言的段 P.18 中所说明的，接口可以派生于多个接口，虽然类只能从一个类派生。

 **程序设计技巧：**当绑定泛型时，使用声明了公有方法 `clone` 的接口，而不是使用 `Cloneable`。不过新接口必须派生于 `Cloneable`。

J9.9

使用 `Copyable` 作为 `T` 的上界，需要我们修改 `AList` 构造方法的实现。回忆第 11 章程序清单 11-1，`AList` 的一个域是线性表项的数组：

```
private T[] list;
```

在构造方法中写语句

```
T[] tempList = (T[]) new Object[initialCapacity + 1];
```

会引起 `ClassCastException` 异常。相反我们写如下的语句

```
T[] tempList = (T[]) new Copyable[initialCapacity + 1]; // Change Object to
// Copyable
```

J9.10

现在可以实现 `clone` 了。在 `try` 块内将调用 `super.clone()`，但其他的任务在 `catch` 块后执行。所以，`AList` 的 `clone` 方法的框架如下：

```
public Object clone()
{
    AList<T> theCopy = null;
    try
    {
        @SuppressWarnings("unchecked")
        AList<T> temp = (AList<T>) super.clone();
        theCopy = temp;
    }
    catch (CloneNotSupportedException e)
    {
        throw new Error(e.toString());
    }
    <For a deep copy, we need to do more here, as you will see. >
    .
    .
    return theCopy;
} // end clone
```

这个方法先调用 `super.clone`，将返回的对象转型为 `AList<T>`。为执行深拷贝，需要克隆是或可能是可变对象的数据域。所以需要克隆数组 `list`。

Java 中的数组有公有的 `clone` 方法；换句话说，它们实现了 `Cloneable`。所以可以在线性表的 `clone` 方法内添加下列语句：

```
theCopy.list = (T[])list.clone();
```

这里不需要 `try` 和 `catch` 块，因为 `clone` 是公有的。

数组的 `clone` 方法为数组中的每个对象创建一个浅拷贝。对于深拷贝，必须克隆每个数组项。因为我们要求，线性表项有公有的 `clone` 方法，所以可以写循环，循环体中含有下列语句：

```
@SuppressWarnings("unchecked")
T temp = (T)list[index].clone();
theCopy.list[index] = temp;
```

可以使用 `index` 和 `AList` 的数据域 `numberOfEntries` 来控制循环，后者记录了线性表中的项数。

所以类 `AList` 的 `clone` 有下列定义：

```
public Object clone()
{
    AList<T> theCopy = null;
    // Clone the list
    try
    {
        @SuppressWarnings("unchecked")
        AList<T> temp = (AList<T>)super.clone();
        theCopy = temp;
    }
    catch (CloneNotSupportedException e)
    {
        throw new Error(e.toString());
    }
    // Clone the list's array
    theCopy.list = list.clone();
    // Clone the entries in the array (list[0] is unused and ignored)
    for (int index = 1; index <= numberOfEntries; index++)
    {
        @SuppressWarnings("unchecked")
        T temp = (T)list[index].clone();
        theCopy.list[index] = temp;
    } // end for
    return theCopy;
} // end clone
```



注：要对可克隆对象的数组 `x` 进行深克隆，可调用 `x.clone()`，然后克隆数组内的每个对象。例如，如果 `myArray` 是 `Thing` 对象的数组，且 `Thing` 实现了 `Cloneable`，则可以写

```
Thing[] clonedArray = (Thing[])myArray.clone();
for (int index = 0; index < myArray.length; index++)
    clonedArray[index] = (Thing)myArray[index].clone();
```

克隆一个链表

现在假定我们想为 ADT 线性表的链式实现，如第 12 章的 `LList` 类或第 18 章的 `Linked-ChainBase` 类，添加 `clone` 方法。（用于这些类的 `clone` 方法几乎是相同的。）给定接口

`Copyable`, 可以用段 J9.8 中给出的一种方式来写类的开头。最终, 类及线性表中的对象必须实现接口 `Cloneable`。所以, 比如 `LList`, 可能有如下的开头:

```
public class LList<T extends Copyable> implements CloneableListInterface<T>
{
    private Node firstNode; // Reference to first node of chain
    private int numberofEntries;
    ...
}
```

`clone` 方法的第一部分, 除了将 `AList` 替换为 `LList` 之外, 可能与段 J9.10 中见过的代码一样。如果仅调用 `super.clone()`, 则方法可能会得到线性表的浅拷贝, 如图 JI9-6 所示。换句话说, 原始线性表和它的克隆都指向同一个结点链表, 这些结点将指向一个数据集。

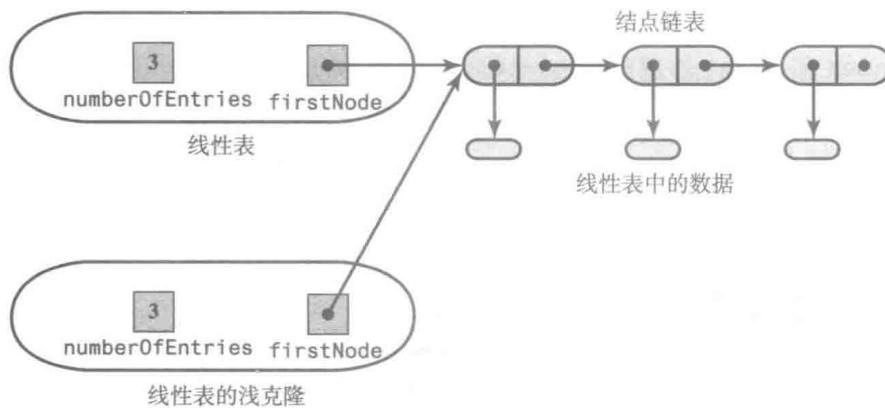


图 JI9-6 线性表和其浅克隆: 链式实现

像以前一样, 要执行深拷贝, `clone` 需要做得更多。它需要克隆结点链表, 也要克隆结点指向的数据。图 JI9-7 显示有深克隆的线性表。

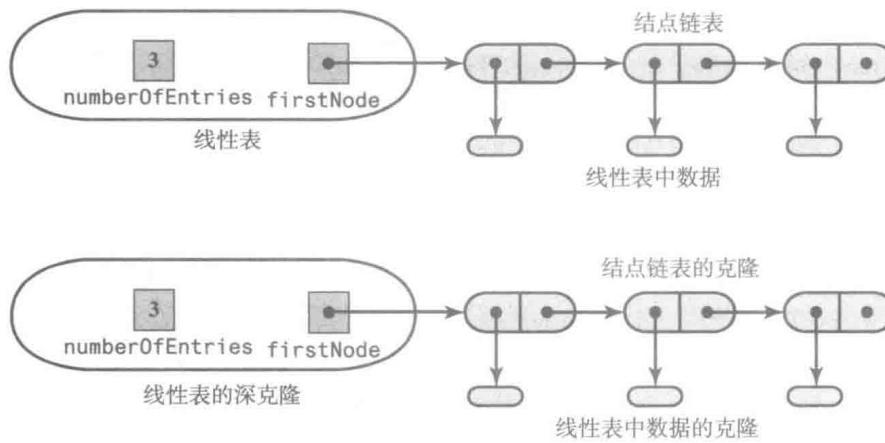


图 JI9-7 线性表和其深克隆: 链式实现

J9.13 克隆一个结点。要克隆链表中的结点, 必须给内部类 `Node` 添加方法 `clone`。首先, 在类 `Node` 的声明中添加 `implements Cloneable`。注意到, `Node` 是 `LList` 中的私有类, 但在 `LinkedChainBase` 中是保护的。`Node` 的 `clone` 方法的开头部分很像是其他的 `clone` 方法, 不过后面它又继续克隆结点的数据部分。我们不必费心来克隆链接, 因为线性表的 `clone` 方法会设置这些。有了这些修改, `LList` 中改版后的 `Node` 如下所示 (修改的部分已标出):

```

private class Node implements Cloneable
{
    private T data;
    private Node next;
    <构造方法>

    <访问方法和赋值方法getData, setData, getNextNode和setNextNode。>
    . . .

    protected Object clone()
    {
        Node theCopy = null;
        try
        {
            @SuppressWarnings("unchecked")
            Node temp = (Node)super.clone();
            theCopy = temp;
        }
        catch (CloneNotSupportedException e)
        {
            throw new Error(e.toString());
        }
        @SuppressWarnings("unchecked")
        T temp = (T)data.clone();
        theCopy.data = temp;
        theCopy.next = null; // Don't clone link; it's set later
        return theCopy;
    } // end clone
} // end Node

```

记住，`data` 调用了不抛出异常的公有 `clone` 方法，所以 `data.clone()` 可以放在 `try` 块的外面。

克隆链表。在类似于下面这样的语句中

J9.14

```
LLList<T> theCopy = (LLList<T>)super.clone();
```

`LLList` 的 `clone` 方法调用了 `super.clone()`。然后方法必须克隆保存线性表数据的结点链表。为此，方法必须遍历链表，克隆每个结点，相应地链接克隆的结点。我们先克隆首结点，以便能正确地设置数据域 `firstNode`：

```
// Make a copy of the first node
theCopy.firstNode = (Node)firstNode.clone();
```

接下来，遍历链表的其余部分。引用 `newRef` 指向已添加到新链表中的最后结点，而引用 `oldRef` 则记录在原链表中遍历的位置。语句

```
newRef.setNextNode((Node)oldRef.clone()); // Attach cloned node
```

克隆原链表中的当前结点，连同数据一起，然后将克隆链接到新链表的末尾。回忆一下，`Node` 的 `clone` 方法还克隆结点指向的数据。图 JI9-8 展示了原始链表和带有被克隆的首结点的新链表。

下面语句实现了前面的思想，并克隆链表的其他部分：

```
Node newRef = theCopy.firstNode;           // Last node in new chain
Node oldRef = firstNode.getNextNode();     // Next node in old chain
for (int count = 2; count <= numberofEntries; count++)
{
    newRef.setNextNode((Node)oldRef.clone()); // Attach cloned node
```

```

newRef = newRef.getNextNode();           // Update references
oldRef = oldRef.getNextNode();
} // end for

```

图 JI9-8b 显示了克隆其第二个结点后的新链表。

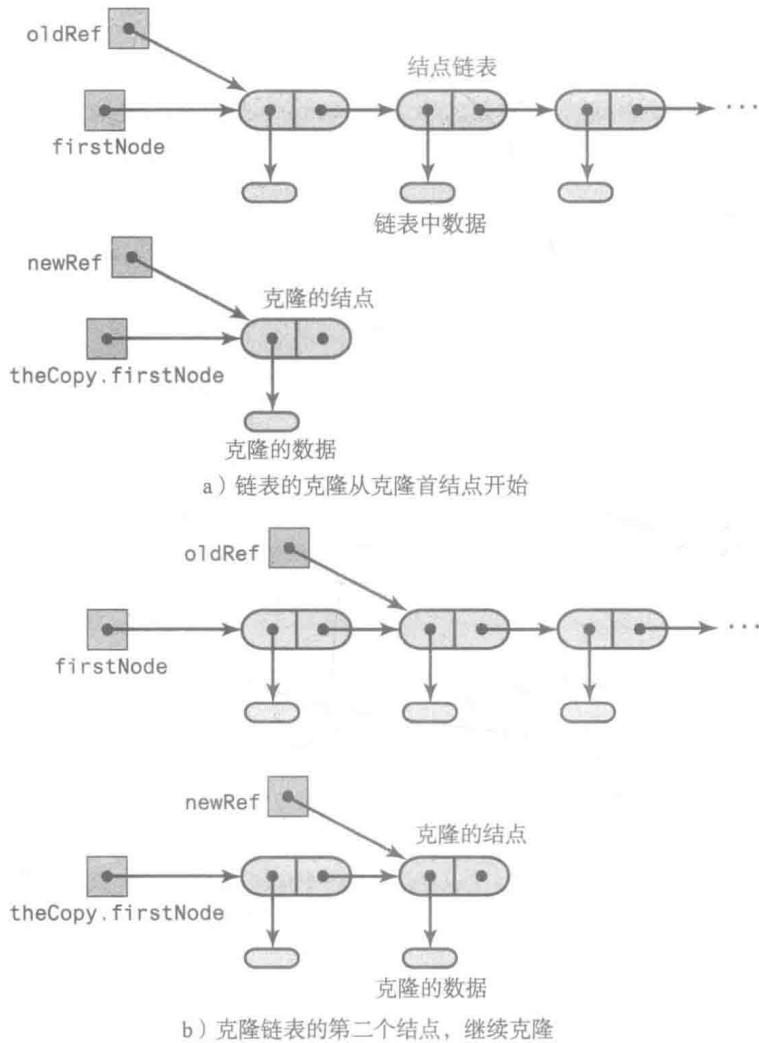


图 JI9-8 结点链表的克隆

J9.15

上面的代码假定处理的是非空结点链表。下面给出的完整 `clone` 方法将检查空链表，并阻止对未检查转型发出警告。

```

public Object clone()
{
    LList<T> theCopy = null;
    try
    {
        @SuppressWarnings("unchecked")
        LList<T> temp = (LList<T>)super.clone();
        theCopy = temp;
    }
    catch (CloneNotSupportedException e)
    {
        throw new Error(e.toString());
    }
}

```

```

// Copy underlying chain of nodes
if (firstNode == null) // If chain is empty
{
    theCopy.firstNode = null;
}
else
{
    // Make a copy of the first node
    @SuppressWarnings("unchecked")
    Node temp = (Node)firstNode.clone();
    theCopy.firstNode = temp;
    // Make a copy of the rest of chain
    Node newRef = theCopy.firstNode;
    Node oldRef = firstNode.getNextNode();
    for (int count = 2; count <= numberOfEntries; count++)
    {
        // Clone node and its data; link clone to new chain
        @SuppressWarnings("unchecked")
        Node temp2 = (Node)oldRef.clone();
        newRef.setNextNode(temp2);
        newRef = newRef.getNextNode();
        oldRef = oldRef.getNextNode();
    } // end for
} // end if
return theCopy;
} // end clone

```



注：对指向可克隆对象的结点链表进行深克隆，必须克隆结点并克隆对象。



安全说明：返回类的数据域的方法，应该返回该域的克隆。



学习问题 2 段 J9.15 中的 for 循环由链表中的结点数控制。修改这个循环，让它由 oldRef 来控制。

有序表的克隆

Java 插曲 6 中的段 J6.4，谈到了将可变对象放在某类集合中的危险，比如有序项组成的线性表这样的集合。如果客户保留了指向任意对象的引用，它就能修改这些对象从而破坏集合的完整性。类似地，对于 ADT 有序线性表实例，客户可能破坏对象的有序性。

段 J6.5 提出了一种办法来解决这个问题，即在集合中只放不可变对象。现在提出另一个解决办法，能让你在集合中放置可变对象。

假定客户向集合中添加了一个对象。假定集合在将对象添加到数据之前克隆了这个对象。然后客户仅能使用 ADT 操作来访问或修改集合的数据。它没有用来修改克隆的指向克隆的引用。当然，这个情况要求，添加的对象是 `Cloneable` 的。我们来研究这样实现 ADT 有序表的细节。

第 17 章段 17.1 说明，有序表中的对象必须是 `Comparable` 的——即它们必须有 `compareTo` 方法。本例中，我们也想让对象是 `Cloneable` 的。段 J9.8 定义了声明公有方法 `clone` 的接口 `Copyable`。使用这个接口，来创建另一个接口：

```

public interface ComparableAndCopyable<T> extends Comparable<T>, Copyable
{

```

J9.16

J9.17

```
    } // end ComparableAndCopyable
```

这个接口能让我们绑定放到有序表中的对象的泛型，下一段中将说明。

实现 `ComparableAndCopyable` 的类必须定义方法 `compareTo` 和 `clone`。例如，可修改段 J9.3 中提到的类 `Name`，在 `clone` 方法之外，还添加方法 `compareTo`，且其开头如下：

```
public class ComparableCopyableName
    implements ComparableAndCopyable<ComparableCopyableName>
```

方法 `clone` 已在段 J9.3 中给出，回答 Java 插曲 5 的学习问题 1 时写过 `compareTo` 方法。

J9.18

因为想让有序表仅含有符合 `Comparable` 和 `Copyable` 的对象，故可以修改用于有序表的接口，如下所示：

```
public interface SortedListOfClonesInterface
    <T extends ComparableAndCopyable<? super T>>
    extends SortedListInterface<T>
{
    } // end SortedListOfClonesInterface
```

然后在类 `LinkedSortedListOfClones` 的定义中使用它：

```
public class LinkedSortedListOfClones
    <T extends ComparableAndCopyable<? super T>>
    implements SortedListOfClonesInterface<T>
```

J9.19

有了解决问题的这些逻辑，现在提出对实现 ADT 有序表的以下修改。可以将这些修改加到第 17 章和第 18 章讨论的实现中：

- 在 `add` 中，将所需项的克隆放到有序表中，而不是放项本身。即将 `newEntry.clone()` 而不是将 `newEntry` 放到表中。所以方法体的开头是这样的

```
Node newNode = new Node((T) newEntry.clone());
```

因为 `clone` 返回 `Object` 的实例，所以转型到泛型 `T` 是必需的。

- 在 `getEntry` 中，返回所需项的克隆而不是项本身。例如，可以返回 `(T) result.clone()` 而不是返回 `result`。

现在更仔细地来讨论这些修改。假定客户有指向对象的引用 `newEntry`，且要将这个对象添加到集合中。集合克隆了对象，然后添加克隆而不是原来的对象，如图 JI9-9 所示。客户没有指向集合中数据的引用。如果客户修改的是 `newEntry` 指向的对象，则集合没有改变。

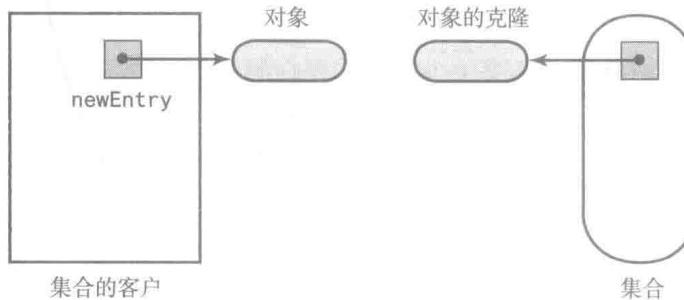


图 JI9-9 将对象的克隆添加到集合之后的集合和它的客户

如果 `getEntry` 没有返回所需项的克隆，而是返回指向集合中所需项的引用该如何呢？如图 JI9-10 所示，客户就能修改集合中的项。所以即使集合含有客户原对象的克隆，`getEntry` 也能让客户访问这个克隆。有了修改这个克隆的能力，客户就可能破坏集合的完

整性。所以，对于 `getEntry`，返回所需项的克隆是必需的。这是客户原始对象克隆的克隆，如图 JI9-11 所示。

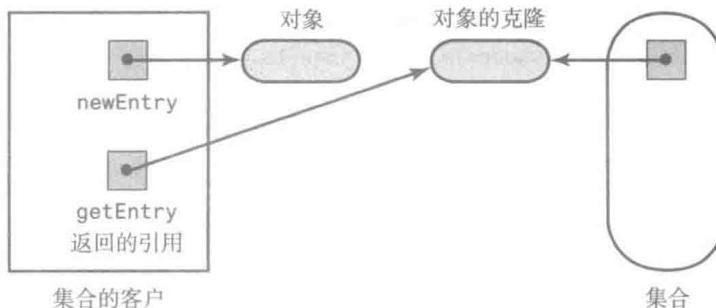


图 JI9-10 如果 `getEntry` 不是返回克隆的后果

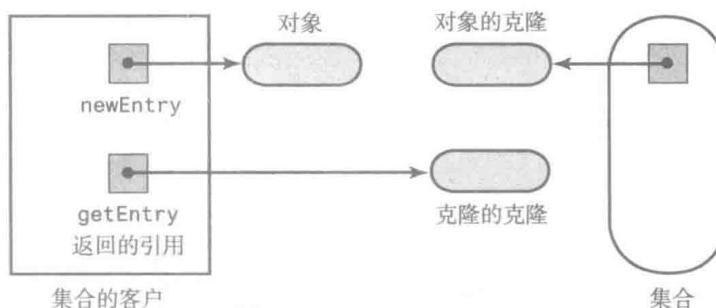


图 JI9-11 当 `getEntry` 返回克隆的后果

 注：集合可以克隆客户添加进来的对象，但你会复制集合中的每个项。对于复杂的对象，每次拷贝所需的时间和内存可能是大量的。

克隆一个二叉结点

第 25 章给出的类 `BinaryNode`，在段 25.5 定义了方法 `copy` 来拷贝一个结点。这个方法也被段 25.7 的类 `BinaryTree` 中的 `initializeTree` 方法使用。现在你应该知道，我们应该定义方法 `clone`，来替代 `copy`。

为此，`BinaryNode` 的 `clone` 方法必须调用 `Object` 的 `clone` 方法，因为 `Object` 是 `BinaryNode` 的超类。然后它必须克隆结点的数据部分，最后克隆结点的两个孩子。下面是得到的方法：

```

/** Makes a clone of this node and its subtrees.
 * @return The clone of the subtree rooted at this node. */
public Object clone()
{
    BinaryNode<T> theCopy = null;
    try
    {
        @SuppressWarnings("unchecked")
        BinaryNode<T> temp = (BinaryNode<T>)super.clone();
        theCopy = temp;
    }
    catch (CloneNotSupportedException e)
    {

```

J9.20

```
    throw new Error("BinaryNode cannot clone: " + e.toString());
}
theCopy.data = (T)data.clone();

if (left != null)
    theCopy.left = (BinaryNode<T>)left.clone();
if (right != null)
    theCopy.right = (BinaryNode<T>)right.clone();

return theCopy;
} // end clone
```

二叉查找树的实现

先修章节：附录 C、第 9 章、第 20 章、第 24 章、第 25 章

目标

学习完本章后，应该能够

- 判定一棵二叉树是否是二叉查找树
- 使用最少的比较次数在二叉查找树中找到给定的项
- 按大小序遍历二叉查找树中的项
- 将新项添加到二叉查找树中
- 从二叉查找树中删除一项
- 描述二叉查找树上操作的效率
- 使用二叉查找树实现 ADT 字典

回忆第 24 章，查找树以一种方便查找的方式存储数据。特别是二叉查找树，它既是二叉树又是查找树。二叉查找树的特性能让我们使用简单的递归算法进行查找。算法的思想类似于数组上的二分查找，也有同样的效率。但是，二叉查找树的树形影响了这个算法的效率。由于从相同的数据能创建几棵不同的二叉查找树，故我们想选择能提供最高查找效率的那个树形的树。

对于比较稳定的数据库来说，二叉查找树提供了能进行高效查找的相对简单的方法。但是大多数数据库都会改变并保存当前的结果，所以我们必须在二叉查找树中添加结点及删除结点。不幸的是，这些操作会改变树的形状，常常使得查找效率变低。

本章实现二叉查找树，为此，要描述添加和删除项的算法。第 28 章介绍有添加及删除操作时仍能保持高效查找的几种查找树。

入门知识

二叉查找树 (binary search tree) 是一棵二叉树，其结点含有 Comparable 类型的对象，并按下列规则组织。对树中的每个结点，

- 结点中的数据大于结点左子树中的数据
- 结点中的数据小于结点右子树中的数据

图 26-1 显示了在第 24 章中见过的一棵二叉查找树。

回忆一下，Comparable 对象属于实现了 Comparable 接口的类。使用类的 `compareTo` 方法来比较这样的对象。依据 `compareTo` 要检测的数据域的不同，各个类进行比较时使用的基准也不同。

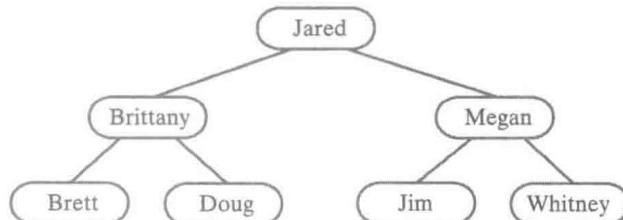


图 26-1 名字的二叉查找树

26.1

用于二叉查找树的接口

26.2 操作。除了接口 TreeInterface 中列出的树的常见操作外，二叉查找树还有基本的数据操作，包括对项的查找、获取、添加、删除和遍历等。我们可以为二叉查找树设计一个接口，这个接口也能用于将在第 28 章中出现的其他查找树。程序清单 26-1 列出了这个接口。注意，查找树中不允许有 null 值，所以，方法可以用返回 null 来表示失败。为了简化讨论，查找树中的项互不相同。

程序清单 26-1 用于查找树的接口

```

1 package TreePackage;
2 import java.util.Iterator;
3 public interface SearchTreeInterface<T extends Comparable<? super T>>
4     extends TreeInterface<T>
5 {
6     /** Searches for a specific entry in this tree.
7      * @param anEntry An object to be found.
8      * @return True if the object was found in the tree. */
9     public boolean contains(T anEntry);
10
11    /** Retrieves a specific entry in this tree.
12     * @param anEntry An object to be found.
13     * @return Either the object that was found in the tree or
14     *         null if no such object exists. */
15    public T getEntry(T anEntry);
16
17    /** Adds a new entry to this tree, if it does not match an existing
18     * object in the tree. Otherwise, replaces the existing object with
19     * the new entry.
20     * @param anEntry An object to be added to the tree.
21     * @return Either null if anEntry was not in the tree but has been added, or
22     *         the existing entry that matched the parameter anEntry
23     *         and has been replaced in the tree. */
24    public T add(T anEntry);
25
26    /** Removes a specific entry from this tree.
27     * @param anEntry An object to be removed.
28     * @return Either the object that was removed from the tree or
29     *         null if no such object exists. */
30    public T remove(T anEntry);
31
32    /** Creates an iterator that traverses all entries in this tree.
33     * @return An iterator that provides sequential and ordered access
34     *         to the entries in the tree. */
35    public Iterator<T> getInorderIterator();
36 } // end SearchTreeInterface

```

26.3 理解规范说明。这些规范说明允许用二叉查找树来实现 ADT 字典，本章稍后会看到。方法使用返回值而不是异常来表示操作是否失败。不过，当获取、添加或删除操作成功时，方法的返回值第一眼看上去可能有些怪异。例如，获取操作 getEntry 返回的值，看上去与让它去查找的项是一样的。但实际上，getEntry 返回的是树中的一个对象，根据项的 compareTo 方法这个对象与所给项是相等的。现在来看一个示例，先添加项然后获取它们。

假定类 Person 有两个字符串类型的数据域，分别表示人的名字和识别号。这个类实现了 Comparable 接口，所以类有 compareTo 方法。假定 compareTo 方法仅比较名字域。考虑下列语句，创建一个对象并添加到二叉查找树中：

```
SearchTreeInterface<Person> myTree = new BinarySearchTree<>();
Person whitney = new Person("Whitney", "111223333");
Person returnValue = myTree.add(whitney);
```

add 操作之后, returnValue 的值是 null, 因为 whitney 还没在树中。现在假定试图添加另一个 Whitney, 他有另外一个识别号:

```
Person whitney2 = new Person("Whitney", "444556666");
returnValue = myTree.add(whitney2);
```

因为 whitney 和 whitney2 有相同的名字, 故它们是相等的, 即表达式 whitney.compareTo(whitney2) 的值是 0。所以, add 方法不会简单地将 whitney2 添加到树中。相反, 它用 whitney2 替换掉 whitney, 并返回树中原来的对象 whitney, 如图 26-2 所示。可以将这个动作看作修改名叫 Whitney 的人的识别号的一种办法。

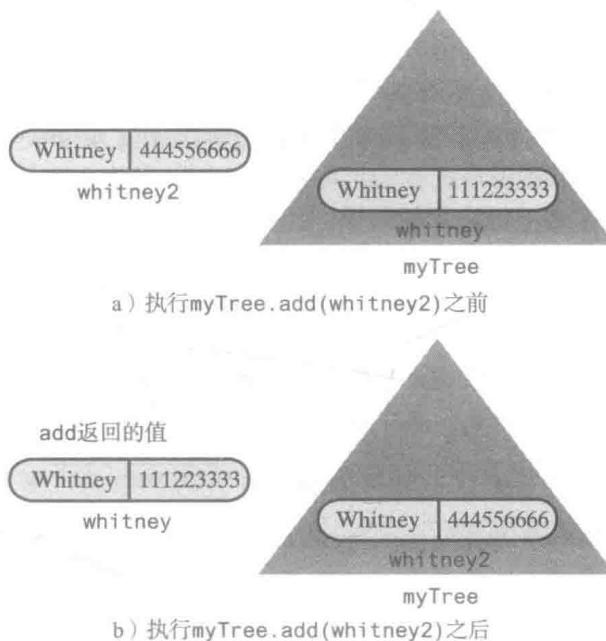


图 26-2 添加一个与二叉查找树中已有项相匹配的项

现在, 语句

```
returnValue = myTree.getEntry(whitney);
```

将 whitney2 赋给 returnValue, 因为 whitney2 已在树中, 且与 whitney 相等。类似地

```
returnValue = myTree.remove(whitney);
```

返回并删除 whitney2。

现在假定方法 compareTo 对 Person 对象的名字和识别号两个域都进行比较。因为根据该 compareTo 方法, whitney 和 whitney2 不相等, 所以可以将两个对象都添加到树中。然后 getEntry(whitney) 将返回 whitney, 而 remove(whitney) 将删除并返回 whitney。

重复项

注意到, SearchTreeInterface 中规范说明的 add 方法能确保树中永远不会添加重复

的值。实际上，这个限制符合大多数应用的需求，但有时也会有例外。可以对二叉查找树的定义做个小小的修改，允许有重复项的存在，即多个项经 `compareTo` 比较是相等的。

图 26-3 显示了一棵二叉查找树，其中 Jared 出现了两次。如果我们处于树根的位置，想知道 Jared 是否再次出现，那么知道应该去哪棵子树进行查看还是有好处的。所以，如果项 e 有重复项 d，则规定 d 出现在 e 所在结点的右子树中。因此，修改定义如下：

对二叉查找树中的每个结点，

- 结点中的数据大于结点左子树中的数据
- 结点中的数据小于或等于结点右子树中的数据

注意到，中序遍历图 26-3 所示的树，访问原有的项 Jared 后立即访问重复的项 Jared。

由于允许有重复项，故 `add` 方法更简单了。但哪个项才是 `getEntry` 要获取的？方法 `remove` 将删除项的首次出现还是所有的出现？到底会出现什么结果要依赖于类的设计者，但这些问题已经说明了重复项带来的复杂性。我们不再深入考虑重复项的问题，而是将这些问题留作你的程序设计项目。(见本章最后的项目 1。)



注：重复项

如果允许在二叉查找树中有重复项，则可以规定重复项在项的右子树中。一旦你选择了右子树，则必须保持一致。本章最后的项目 2 有另外一种重复项的处理机制。



学习问题 1 如果将重复项 Megan 添加到图 26-3 所示的二叉查找树中作为叶结点，则这个新结点应该放在哪里？

开始定义类

26.5

类的框架。让我们开始定义二叉查找树的类。因为二叉查找树是一棵二叉树，所以从第 25 章定义的类 `BinaryTree` 派生新类。现在开始写我们的类，如程序清单 26-2 所示。注意，调用保护方法 `setRootNode` 的构造方法，它的类继承于 `BinaryTree` 类。第 25 章段 25.8 中有 `setRootNode` 的定义。

程序清单 26-2 类 `BinarySearchTree` 的框架

```

1 package TreePackage;
2
3 public class BinarySearchTree<T extends Comparable<? super T>>
4     extends BinaryTree<T>
5     implements SearchTreeInterface<T>
6 {
7     public BinarySearchTree()
8     {
9         super();
10    } // end default constructor
11
12    public BinarySearchTree(T rootEntry)
13    {

```

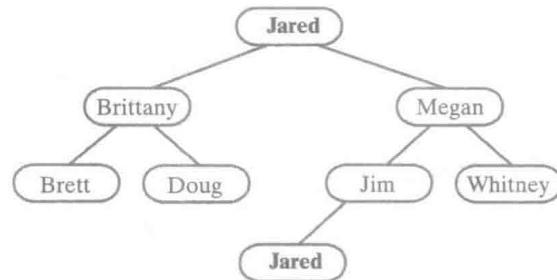


图 26-3 有重复项的二叉查找树

```

14     super();
15     setRootNode(new BinaryNode<T>(rootEntry));
16 } // end constructor
17
18 // Disable setTree (see Segment 26.6)
19 public void setTree(T rootData, BinaryTreeInterface<T> leftTree,
20                      BinaryTreeInterface<T> rightTree)
21 {
22     throw new UnsupportedOperationException();
23 } // end setTree
24
25 < Implementations of contains, getEntry, add, and remove are here. Their definitions appear
26 in subsequent sections of this chapter. Other methods in SearchTreeInterface are inherited
27 from BinaryTree. >
28 ...
29 } // end BinarySearchTree

```



注：类 `BinarySearchTree` 不是终极类，所以可以将其用作基类。

让 `setTree` 方法失效。进行其他讨论之前，先来考虑继承于 `BinaryTree` 类的 `setTree` 方法。客户可能使用这个方法来创建一棵树，不幸的是，创建的不是一棵二叉查找树。如果客户使用 `SearchTreeInterface` 来声明树的实例，就不可能是这样的结果。例如，如果写语句

```
SearchTreeInterface<String> dataSet = new BinarySearchTree<String>();
```

则 `dataSet` 就没有这个 `setTree` 方法，因为它不在 `SearchTreeInterface` 中。但如果我们将写

```
BinarySearchTree<String> dataSet = new BinarySearchTree<String>();
```

则 `dataSet` 就有这个 `setTree` 方法。

为避免客户使用 `setTree` 方法，我们重写它，如果被调用则抛出一个异常。程序清单 26.2 显示的是这种思路下 `setTree` 方法的定义。



学习问题 2 `BinarySearchTree` 类中第二个构造方法调用方法 `setRootNode`。能不能用调用 `setRootData(rootEntry)` 来代替这个调用？给出解释。

学习问题 3 有没有必要在类 `BinarySearchTree` 中定义方法 `isEmpty` 和 `clear`？给出解释。

查找和获取

查找算法。 第 24 章的段 24.30 给出了下列在一棵二叉查找树中进行查找的递归算法：

26.7

```

Algorithm bstSearch(binarySearchTree, desiredObject)
// Searches a binary search tree for a given object.
// Returns true if the object is found.

if (binarySearchTree 为空)
    return false
else if (desiredObject == binarySearchTree根中的对象)
    return true
else if (desiredObject < binarySearchTree根中的对象)
    return bstSearch(binarySearchTree的左子树, desiredObject)

```

```

else
    return bstSearch(binarySearchTree的右子树, desiredObject)

```

这个算法是方法 `getEntry` 的基础。

 **注：**在二叉查找树中的查找很像在数组上的二分查找：查找二叉查找树中两棵子树的一棵，而不是查找数组的一半。

26.8 虽然用树和子树很容易表示递归算法，但第 25 章二叉树的实现中暗示使用的是根结点。树或子树的根结点给我们提供了一种方法，可用来查找或操作其后代结点。

下面的算法与上面刚给出的算法等价，但所描述的更接近于实际的实现。

```

Algorithm bstSearch(binarySearchTreeRoot, desiredObject)
// Searches a binary search tree for a given object.
// Returns true if the object is found.

if (binarySearchTreeRoot 等于 null)
    return false
else if (desiredObject == binarySearchTreeRoot 中的对象)
    return true
else if (desiredObject < binarySearchTreeRoot 中的对象)
    return bstSearch(binarySearchTreeRoot 的左孩子, desiredObject)
else
    return bstSearch(binarySearchTreeRoot 的右孩子, desiredObject)

```

我们仍用树和子树来表示后面的算法，但在实现中使用根结点但不明确地提到它。

26.9 **方法 `getEntry`**。就像通常的递归算法那样，我们将实际的查找过程实现为一个私有方法 `findEntry`，而由公有方法 `getEntry` 来调用。虽然算法返回一个逻辑值，但我们的实现将返回找到的数据对象。故方法如下：

```

public T getEntry(T anEntry)
{
    return findEntry(getRootNode(), anEntry);
} // end getEntry

private T findEntry(BinaryNode<T> rootNode, T anEntry)
{
    T result = null;
    if (rootNode != null)
    {
        T rootEntry = rootNode.getData();
        if (anEntry.equals(rootEntry))
            result = rootEntry;
        else if (anEntry.compareTo(rootEntry) < 0)
            result = findEntry(rootNode.getLeftChild(), anEntry);
        else
            result = findEntry(rootNode.getRightChild(), anEntry);
    } // end if
    return result;
} // end findEntry

```

我们使用方法 `compareTo` 和 `equals` 来比较给定的项与树中已有的项。还要注意用到了 `BinaryNode` 类中的方法。我们假定对这个类至少有包访问权限。

也可以用迭代方式实现方法 `getEntry`，可用也可不用像 `findEntry` 这样的私有方法。将这个实现留作练习。

26.10 方法 `contains`。方法 `contains` 可以简单地调用 `getEntry`，来看看给定的项是否在树中：

```

public boolean contains(T anEntry)
{
    return getEntry(anEntry) != null;
} // end contains

```

遍历

SearchTreeInterface 提供了方法 `getInorderIterator`，它返回一个中序迭代器。26.11 因为我们的类是 `BinaryTree` 的子类，所以它继承了 `getInorderIterator` 方法。对于一棵二叉查找树，这个迭代器按升序遍历各项，这由项的方法 `compareTo` 来定义。



学习问题 4 如果想让指向 `BinarySearchTree` 对象的引用能调用 `TreeIterator-Interface` 中的其他方法，该如何声明这个引用？

添加一项

26.12 在二叉查找树中添加一项是个基本操作，因为那就是我们初始时创建树的方法。假定有一棵二叉查找树，现在要将一个新项添加到树中。我们不能将它随随便便地添加在树的某个地方，因为必须要保持结点之间的关系。也就是说，添加后树必须仍是一棵二叉查找树。另外，方法 `getEntry` 必须能找到这个新项。例如，如果想将项 Chad 添加到图 26-4a 所示的树中，就不能将新结点添加到 Jared 的右子树中。因为 Chad 排在 Jared 的前面，Chad 必须在 Jared 的左子树中。因为 Brittany 是那棵左子树的根，所以比较 Chad 与 Brittany，发现 Chad 更大。因此 Chad 属于 Brittany 的右子树。继续这个过程，比较 Chad 与 Doug，发现 Chad 属于 Doug 的左子树。但这棵树是空的，即 Doug 没有左孩子。

如果让 Chad 成为 Doug 的左孩子，则得到图 26-4b 所示的二叉查找树。现在 `getEntry` 通过刚刚描述的那些比较，能够找到 Chad。也就是说，`getEntry` 在找到 Chad 之前，将 Chad 与 Jared、Brittany 和 Doug 依次进行比较。注意，新结点是一个叶结点。



注：二叉查找树的每次添加都是增加了树的一个叶结点。

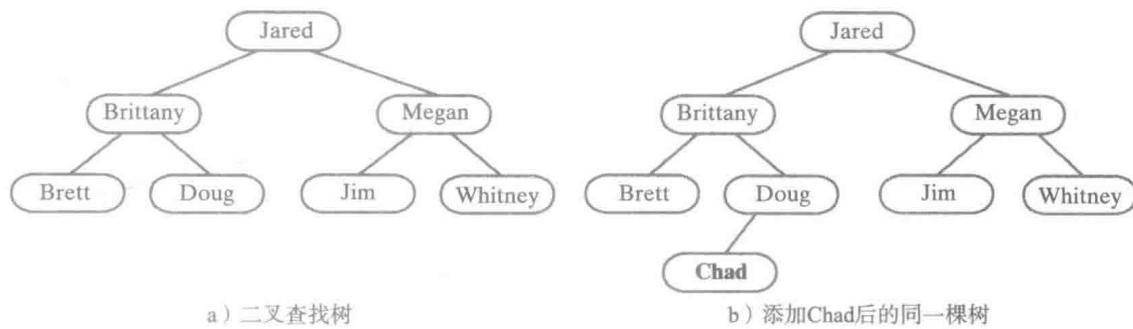


图 26-4 添加 Chad 之前和之后的二叉查找树

学习问题 5 将名字 Chris、Jason 和 Kelley 添加到图 26-4b 所示的二叉查找树中。

学习问题 6 将名字 Miguel 添加到图 26-4a 所示的二叉查找树中，然后添加 Nancy。现在回到原来的树中，添加 Nancy，然后添加 Miguel。添加两个名字的次序影响得到的树的结果吗？

递归实现

26.13

方法 add 有一个简洁的递归实现版本。再次考虑前一段中给出的示例。图 26-5 展示了将 Chad 递归地添加到图 26-4a 所示的二叉查找树中的步骤：

- 要将 Chad 添加到根为 Jared 的二叉查找树中，如图 26-5a 所示：
观察 Chad 小于 Jared。
将 Chad 添加到 Jared 的左子树中，其根为 Brittany，如图 26-5b 所示。
- 要将 Chad 添加到根为 Brittany 的二叉查找树中：
观察 Chad 大于 Brittany。
将 Chad 添加到 Brittany 的右子树中，其根为 Doug，见图 26-5c。
要将 Chad 添加到根为 Doug 的二叉查找树中：
观察 Chad 小于 Doug。
因为 Doug 没有左子树，故让 Chad 成为 Doug 的左孩子。

图 26-5d 展示了添加的结果。可以看到，将一个项添加到以 Jared 为根的树中，依赖于将其添加到后续更小的子树中。

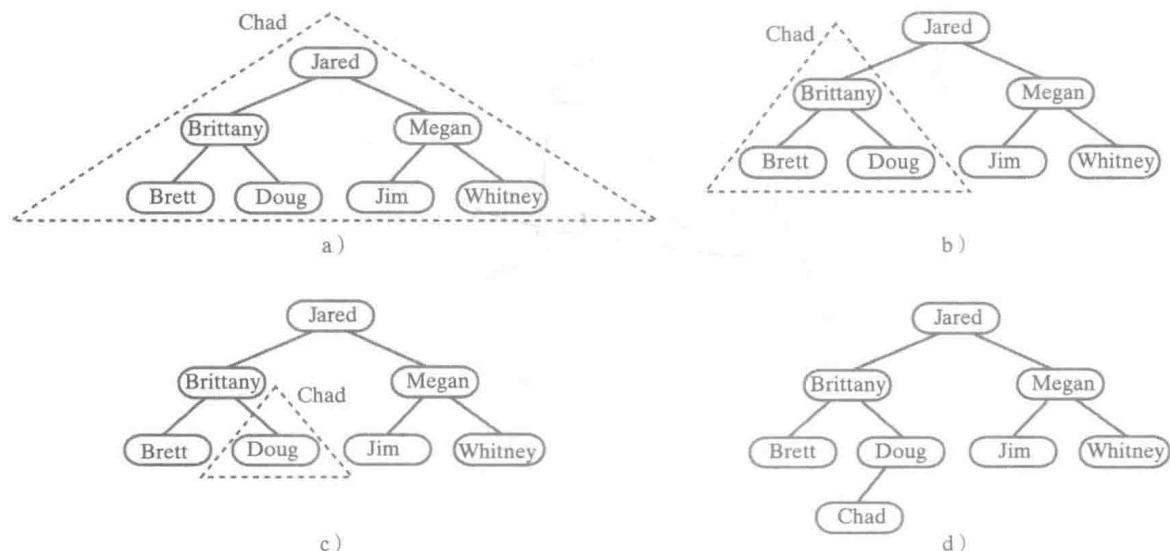


图 26-5 递归地将 Chad 添加到二叉查找树更小的子树中

26.14

添加新项的递归算法。根据 `SearchTreeInterface` 中 `add` 方法的规范说明，将这个方法形式化描述为如下的递归算法。回忆一下，我们决定在二叉查找树中各项均不相同。如果试图添加到树中的项与树中已有的项相等，则用新项进行替换，并返回原来的项。

为简化算法，假定目前二叉查找树不为空：

```

Algorithm addEntry(binarySearchTree, anEntry)
// Adds an entry to a binary search tree that is not empty.
// Returns null if anEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by anEntry.

result = null
if (anEntry 等于 binarySearchTree 的根中的项)
{
    result = 根中的项
    用 anEntry 替换根中的项
}

```

```

}
else if (anEntry < binarySearchTree 的根中的项)
{
    if (binarySearchTree 的根有左孩子)
        result = addEntry(binarySearchTree 的左子树, anEntry)
    else
        将含有 anEntry 的结点赋给根做左孩子
}
else // anEntry > entry in the root of binarySearchTree
{
    if (binarySearchTree 的根有右孩子)
        result = addEntry(binarySearchTree 的右子树, anEntry)
    else
        将含有 anEntry 的结点赋给根做右孩子
}
return result

```

将项添加到空二叉查找树中可以作为特例，在调用 `addEntry` 的另一个算法中进行另外的处理，如下所示。

```

Algorithm add(binarySearchTree, anEntry)
// Adds an entry to a binary search tree.
// Returns null if anEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by anEntry.

result = null
if (binarySearchTree 为空)
    创建含有 anEntry 的结点，让它成为 binarySearchTree 的根
else
    result = addEntry(binarySearchTree, anEntry)
return result;

```

私有的递归方法 addEntry。 回忆一下，段 26.7 给出了递归查找算法。段 26.9 中的公有方法 `getEntry` 调用一个实现查找算法的私有递归方法 `findEntry`。这里的情况类似。如果树不为空，则公有方法 `add` 调用私有的递归方法 `addEntry`。与 `findEntry` 一样，`addEntry` 有一个结点作为形参，初始时是树的根结点。当递归调用 `addEntry` 时，这个参数或者是当前根结点的左孩子，或者是其右孩子。26.15

如图 26-4 和图 26-5 所示，将新项放到要添加到二叉查找树的新叶结点中。现在设想一下，当将 Chad 添加到图 26-5a 所示的树中时递归调用 `addEntry` 的过程。最后，含有 Doug 的结点作为实参传给 `addEntry` (图 26-5c)。因为 Chad 小于 Doug，而 Doug 所在的结点没有左孩子，所以 `addEntry` 创建了一个包含 Chad 的结点 (图 26-5d)。

`addEntry` 的下列实现严格遵从段 26.14 中给出的伪代码。

```

// Adds anEntry to the nonempty subtree rooted at rootNode.
private T addEntry(BinaryNode<T> rootNode, T anEntry)
{
    // Assertion rootNode != null
    T result = null;
    int comparison = anEntry.compareTo(rootNode.getData());
    if (comparison == 0)
    {
        result = rootNode.getData();
        rootNode.setData(anEntry);
    }
    else if (comparison < 0)
    {
        if (rootNode.hasLeftChild())

```

```

        result = addEntry(rootNode.getLeftChild(), anEntry);
    else
        rootNode.setLeftChild(new BinaryNode<>(anEntry));
    }
else
{
    // Assertion comparison > 0
    if (rootNode.hasRightChild())
        result = addEntry(rootNode.getRightChild(), anEntry);
    else
        rootNode.setRightChild(new BinaryNode<>(anEntry));
} // end if
return result;
} // end addEntry

```

从将新项与根中的项进行比较开始。如果项是相等的，则替换并返回根中原来的项。如果比较结果是“小于”，且根有左孩子，则将那个孩子传递给 `addEntry`。记住当编写像 `addEntry` 这样的递归方法时，写递归调用时假定它是有效的。所以 `addEntry` 将含有 `addEntry` 的新结点放入根的左子树中。如果根没有左孩子，则将含有新项的结点作为其左孩子。当新项大于根中的项时的处理与此类似。

26.16 公有方法 add。公有方法 `add` 不仅要调用递归方法 `addEntry`，还要确保传给 `addEntry` 的根不能是空的。因此，`add` 自己处理空树。遵循段 26.14 中所给的算法，`add` 的实现如下。注意继承于类 `BinaryTree` 的保护方法 `setRootNode` 和 `getTreeNode` 的使用。

```

public T add(T anEntry)
{
    T result = null;
    if (isEmpty())
        setRootNode(new BinaryNode<>(anEntry));
    else
        result = addEntry(getRootNode(), anEntry);
    return result;
} // end add

```

迭代实现

可以迭代实现 `addEntry` 方法。模仿前面给出的 `addEntry` 的递归版本的逻辑，因此你可以将两个方法做个对比。本章最后的练习 12 要求实现另一个迭代算法。

26.17 添加新项的迭代算法。下列迭代算法将新项添加到一棵非空二叉查找树中。

```

Algorithm addEntry(binarySearchTree, anEntry)
// Adds a new entry to a binary search tree that is not empty.
// Returns null if anEntry did not exist already in the tree. Otherwise, returns the
// tree entry that matched and was replaced by anEntry.

result = null
currentNode = binarySearchTree 的根结点
found = false

while (found 为假)
{
    if (anEntry 等于 currentNode 中的项)
    {
        found = true
        result = currentNode 中的项
        用 anEntry 替换 currentNode 中的项
    }
    else if (newEntry < currentNode 中的项)

```

```

{
    if (currentNode有左孩子)
        currentNode = currentNode的左孩子
    else
    {
        found = true
        将含有anEntry的结点作为currentNode的左孩子
    }
}
else // anEntry > entry in currentNode
{
    if (currentNode有右孩子)
        currentNode = currentNode的右孩子
    else
    {
        found = true
        将含有anEntry的结点作为currentNode的右孩子
    }
}
}
return result

```

while 循环试图将新项与树中已有的项进行匹配。如果新项不在树中，则对它的查找将结束于结点的值为 null 的孩子引用上。这就是要放置新结点的位置。但如果新项与树中的某个项相等，则返回树中已有的项，并用新项替代树中的这个项。

方法 addEntry 的迭代实现。前一个算法的 Java 实现严格遵从算法的逻辑。注意继承于 BinaryTree 类的保护方法 getRootNode 的使用。26.18

```

private T addEntry(T anEntry)
{
    BinaryNode<T> currentNode = getRootNode();
    // Assertion currentNode != null
    T result = null;
    boolean found = false;

    while (!found)
    {
        T currentEntry = currentNode.getData();
        int comparison = anEntry.compareTo(currentEntry);

        if (comparison == 0)
        { // anEntry matches currentEntry;
            // return and replace currentEntry
            found = true;
            result = currentEntry;
            currentNode.setData(anEntry);
        }
        else if (comparison < 0)
        {
            if (currentNode.hasLeftChild())
                currentNode = currentNode.getLeftChild();
            else
            {
                found = true;
                currentNode.setLeftChild(new BinaryNode<T>(anEntry));
            } // end if
        }
        else
        {
            // Assertion comparison > 0
            if (currentNode.hasRightChild())
                currentNode = currentNode.getRightChild();
        }
    }
}

```

```

    else
    {
        found = true;
        currentNode.setRightChild(new BinaryNode<>(anEntry));
    } // end if
} // end if
} // end while
return result;
} // end addEntry

```

除了实际调用的 `addEntry` 方法不同外，调用这个迭代版本 `addEntry` 的方法 `add`，类似于段 26.16 中所给的那个方法。因为迭代版本的 `addEntry` 有一个而不是两个参数，调用时用 `addEntry(anEntry)` 而不是用 `addEntry(getRootNode(), anEntry)`。

! **程序设计技巧：**是使用练习 12 中提出的这个迭代的 `addEntry` 方法，还是前面给出的递归版本，某种程度上要看哪个版本最适合你。如果真的能明白你的算法，调试时花的时间更少。

删除一项

26.19 要从二叉查找树中删除一项，需要将要查找的项传递给方法 `remove`。然后从树中删除要找的这个项，并返回给客户。如果没有这样的项存在，则方法返回 `null`，树保持不变。

删除一个项比添加一个项更难一些，操作逻辑取决于含有项的结点的孩子个数。有 3 种可能：

- 结点没有孩子——它是叶结点
- 结点有一个孩子
- 结点有两个孩子

现在考虑这 3 种情形。

删除叶结点中的项

26.20 删除项时最简单的情形是结点为叶子，即它没有孩子结点。例如，假定结点 N 含有要从二叉查找树中删除的项。图 26-6 显示了结点 N 的两种可能情况：它可能是其父结点 P 的左孩子或右孩子。因为 N 是叶结点，所以将结点 P 中相应的孩子引用置为 `null` 就可以了，如图 26-6 所示。

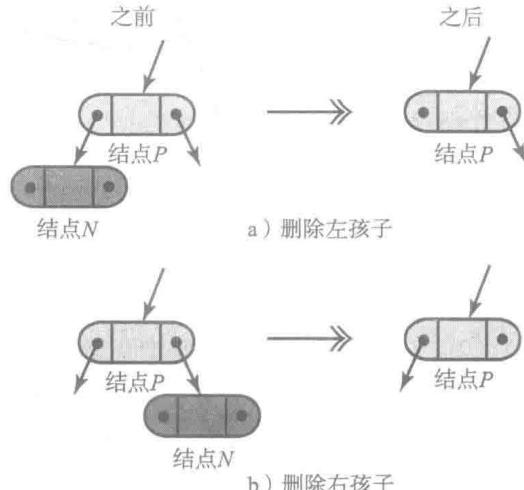


图 26-6 从父结点 P 中删除叶结点 N

删除仅有一个孩子的结点中的项

26.21 现在假定要删除的项位于只有一个孩子 C 的结点 N 中。图 26-7a 显示了结点 N 和其父结点 P 的 4 种可能情况。 N 或者是 P 的左孩子或者是 P 的右孩子，而 C 或者是 N 的左孩子或者是 N 的右孩子。为了删除 N 中的项，要从树中删除 N 。让 C 替代 N 作为 P 的孩子就可以了。如图 26-7b 所示，如果 N 是 P 的左孩子，则让 C 成为 P 的左孩子。类似地，如果 N 是 P 的右孩子，则让 C 成为 P 的右孩子。

得到的树还是二叉查找树吗？例如，如果 N 是 P 的左孩子， P 中的项大于 P 左子树中的所有项。删除 N 后，这个关系仍然是对的，所以树仍然是二叉查找树。当 N 是 P 的右孩子时，情况类似。

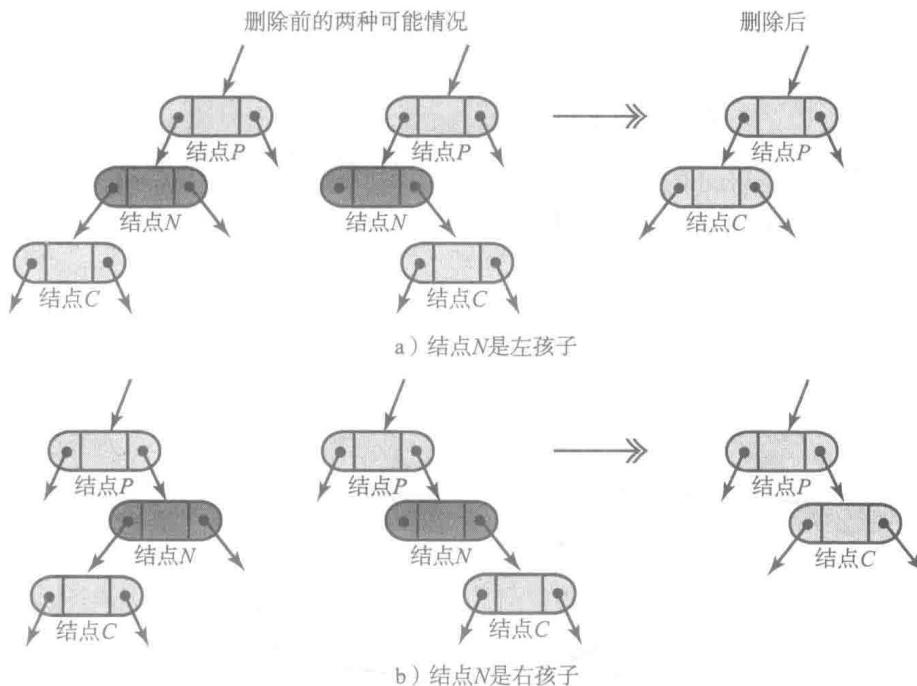


图 26-7 从父结点 P 中删除有一个孩子的结点 N

删除有两个孩子的结点中的项

前两种情况——当 N 有不多于 1 个的孩子——在概念上或实现时都不太难。但最后这种情况有点复杂。再次假定要删除的项在结点 N 中，但现在 N 有两个孩子。图 26-8 显示了 N 的两种可能情况。如果删除了结点 N ，会让它的两个孩子都没有父结点。虽然结点 P 可以指向其中的一个，但它提供不出两个空间。所以删除结点 N 不应是我们的选择。26.22

记住，我们的目标是从树中删除一个项。实际上并不要为了删除项而不得不删除结点 N 。我们去找一个容易删除的结点 X ——它的孩子不会多于 1 个——用 X 中的项来替代 N 中的项。然后可以删除结点 X ，且树中仍有正确的项。但树仍是二叉查找树吗？显然，结点 X 不能是任意的结点；它必须含有一个放到结点 N 处也合适的树中的项。

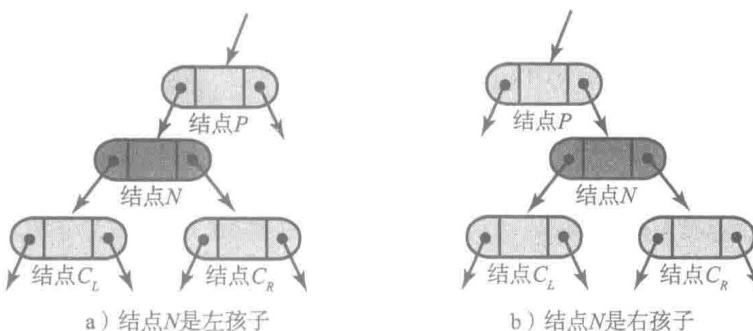


图 26-8 有两个孩子的结点 N 的两种可能情况

26.23 我们知道树中的项互异。令 target 为我们要删除的项；假定它在结点 N 中。因为结点 N 有两个孩子，则 target 大于 N 的左孩子中的项而小于 N 的右孩子中的项。所以，target 不是树中最小的项，也不可能最大的。如果假定树中的项按升序排列，令 pred 是紧邻 target 之前的项，令 succ 是紧邻 target 之后的项。中序遍历这棵树将按升序的次序访问这些项，即 pred、target、succ。所以 pred 称为 target 的中序前驱，而 succ 称为 target 的中序后继。

项 pred 必须出现在 N 的左子树的一个结点中；而 succ 在 N 的右子树的一个结点中，如图 26-9a 所示。进一步，pred 是 N 的左子树中的最大项，因为 pred 是紧邻于 target 之前的项。假设我们能删除含 pred 的结点，并用 pred 来替换 target，如图 26-9b 所示。现在根据需要，N 的左子树中剩余的所有项都小于 pred。N 的右子树中的所有项都大于 target，因此也大于 pred。故仍得到一棵二叉查找树。

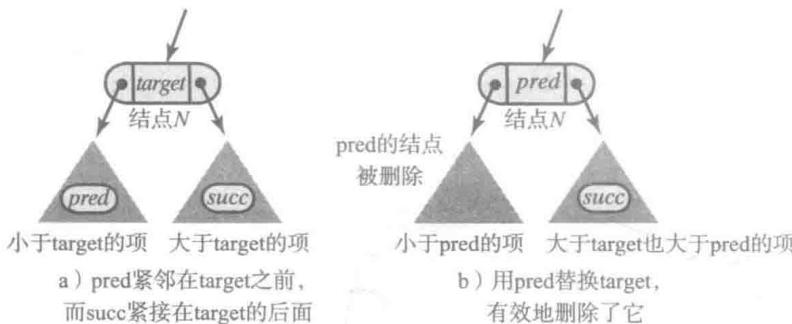


图 26-9 删掉 target 之前和之后，结点 N 和它的子树

26.24 找到项 pred。上一段假定我们能找到 target 的中序前驱 pred，并删除它所在的结点。现在我们来查找含 pred 的结点。再来看图 26-9a 所示的树，从树中删除 target 之前的结点 N。我们已经知道项 pred 必须在 N 的左子树中，且 pred 是那棵子树中最大的项。所以 pred 出现在子树最右的结点 R 处，如图 26-10 所示。结点 R 不能有右孩子，因为如果有，孩子中的项就要大于 pred。故结点 R 的孩子个数不多于 1 个，容易从树中删除。

26.25 下列伪代码概述了刚才的讨论：

```
删除有两个孩子的结点 N 中的项 target 的算法
找到 N 的左子树中的最右结点 R
用结点 R 中的项替换结点 N 中的项
删除结点 R
```

另一个方法涉及 succ，图 26-9a 中有序紧邻在 target 之后的项。已经注意到 succ 出现在 N 的右子树中。它必须是那棵子树中最小的项，所以应该出现在子树最左的结点中。故又得到如下的另一个伪代码：

```
删除有两个孩子的结点 N 中的项 target 的算法
找到 N 的右子树中的最左结点 L
用结点 L 中的项替换结点 N 中的项
删除结点 L
```

两个方法同样有效。

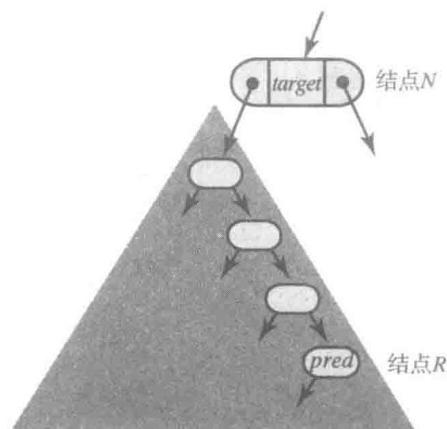


图 26-10 结点 N 左子树中的最大项 pred 出现在子树最右的结点 R 中



注：要删除有两个孩子的结点中的项，首先用另一个最多有1个孩子的结点中的项来替换这个项。然后从树中删除那个最多有一个孩子的结点。



示例。图 26-11 给出的是从名字二叉查找树中连续删除的过程。使用上面伪代码给出的第一个算法。要从图 26-11a 所示的树中删除 Chad，用中序前驱 Brittany 来替代它。然后删除含有 Brittany 的结点，得到图 26-11b 所示的树。要从这棵新树中删除 Sean，用中序前驱 Reba 替代它，并删去 Reba 所在的原结点。这得到了图 26-11c 所示的树。最后从这棵树中删除 Kathy，用其中序前驱 Doug 替代它，并删除 Doug 所在的原结点，得到图 26-11d 所示的树。

26.26

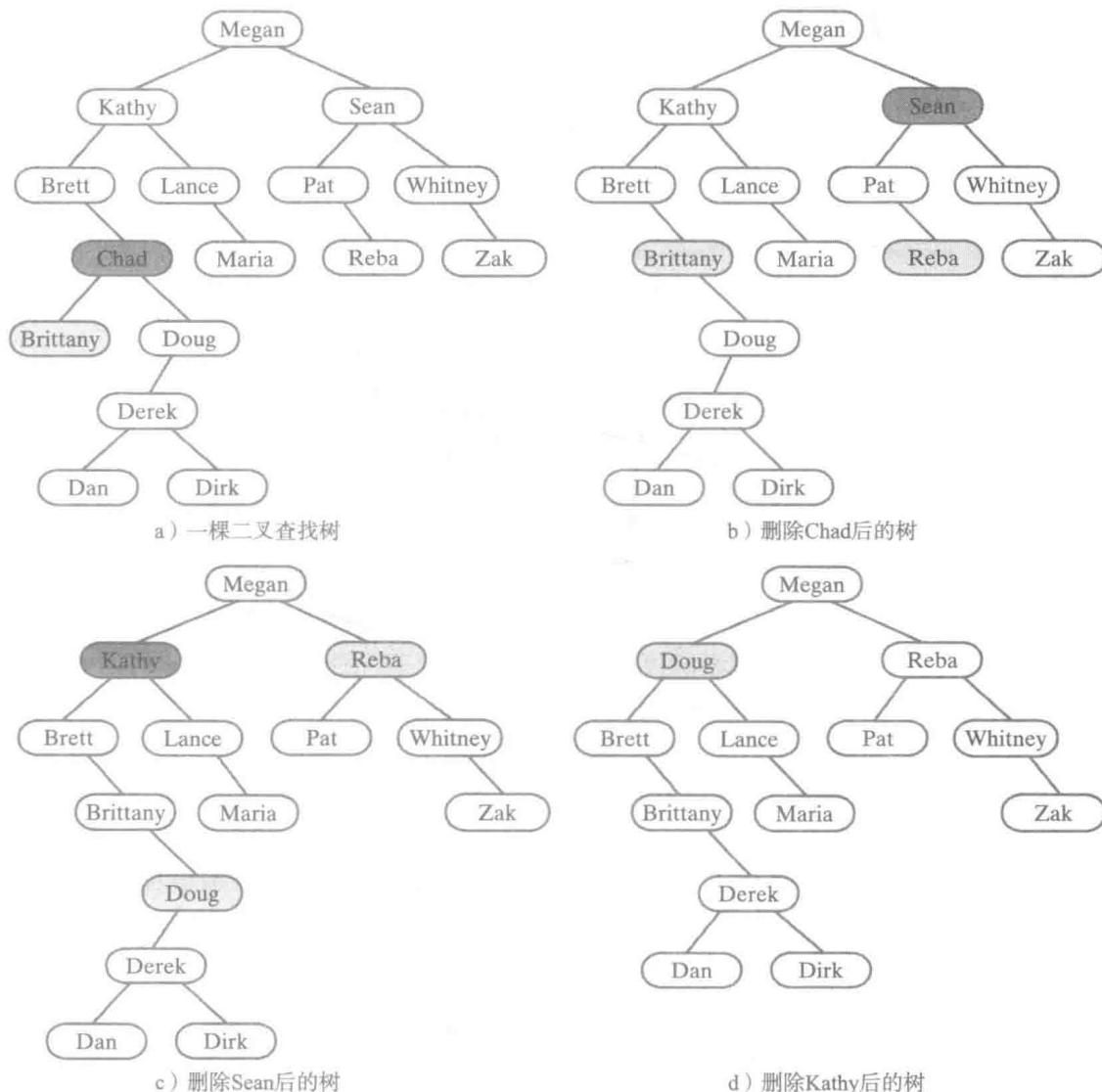


图 26-11 从一棵二叉查找树中连续删除



学习问题 7 段 26.25 中描述的第二个算法涉及中序后继。使用这个算法，从图 26-11a 所示的树中删除 Sean 和 Chad。

学习问题 8 从图 26-11a 所示的树中，使用两种不同的方法删除 Megan。

删除根中的项

26.27

删除树根中的项时会有一个特例，就是真的删除了根结点。当根最多只有一个孩子时就会发生这种情况。如果根有两个孩子，则上面的示例表明我们可以替换掉根中的项并删除另外一个结点。

如果根是叶子，则树只有一个结点，删除它得到一棵空树。如果根有一个孩子，如图 26-12 所示，则孩子或是右孩子或是左孩子。这两种情况下，我们都可以用孩子结点 C 作为树的根从而删除根结点。

递归实现

26.28

算法。要从树中删除的项，是与作为实参传递给方法 `remove` 的对象相匹配的。方法返回被删除的项。下列递归算法抽象地描述了方法的逻辑：

```
Algorithm remove(binarySearchTree, anEntry)
oldEntry = null
if (binarySearchTree 不为空)
{
    if (anEntry 等于 binarySearchTree 根中的项)
    {
        oldEntry = 根中的项
        removeFromRoot(binarySearchTree 的根)
    }
    else if (anEntry < 根中的项)
        oldEntry = remove(binarySearchTree 的左子树, anEntry)
    else // anEntry > entry in root
        oldEntry = remove(binarySearchTree 的右子树, anEntry)
}
return oldEntry
```

方法 `removeFromRoot` 根据根的孩子个数，删除给定子树的根中的项。

26.29

公有方法 remove。在实现前一个算法之前还有几个细节要考虑。公有方法 `remove` 应该仅有一个形参——`anEntry`——所以与调用私有递归方法 `addEntry` 的方法 `add` 一样，`remove` 将调用一个私有递归方法 `removeEntry`。

正如我们在段 26.8 中提到过的，给 `removeEntry` 传递的是树根，而不是树本身。因为方法可能从树中删除根结点，故我们必须谨慎，总要保留指向树根的引用。因此，让 `removeEntry` 返回的也是指向新树根的引用，它可由 `remove` 保存。不过，`removeEntry` 还必须将它删除的项也传给 `remove`。一种办法是给 `removeEntry` 再传另外一個形参——`oldEntry`，然后在方法中用删除的项来改变它的值。所以，`removeEntry` 的头会是这样的：

```
private BinaryNode<T> removeEntry(BinaryNode<T> rootNode, T anEntry,
                                    ReturnObject oldEntry)
```

`ReturnObject` 是一个内部类，仅有一个数据域，只有方法 `set` 和 `get` 可使用。初始时，`oldEntry` 的数据域是 `null`，因为当在树中没有找到项时，`remove` 返回 `null`。

所以，公有的 `remove` 方法的实现如下：

```
public T remove(T anEntry)
{
```

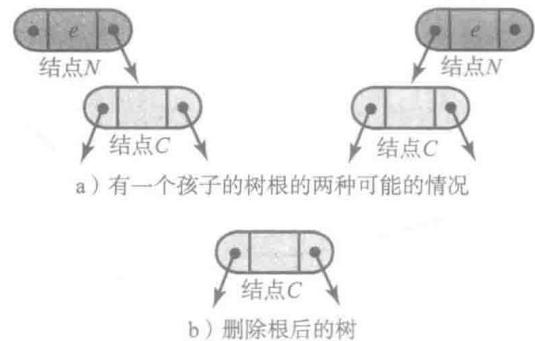


图 26-12 当根只有一个孩子时删除根

```

ReturnObject oldEntry = new ReturnObject(null);
BinaryNode<T> newRoot = removeEntry(getRootNode(), anEntry, oldEntry);
setRootNode(newRoot);
return oldEntry.get();
} // end remove

```

私有方法 `removeEntry`。因为 `remove` 调用 `removeEntry`，故段 26.28 中算法的大部分工作留给 `removeEntry` 来处理。如果要删除的项在根中，则 `removeEntry` 调用尚未编写的方法 `removeFromRoot` 来删除它。如果项在根的一棵子树中，则 `removeEntry` 递归调用它自己。`removeEntry` 的实现如下所示。

```

// Removes an entry from the tree rooted at a given node.
// Parameters:
//   rootNode A reference to the root of a tree.
//   anEntry The object to be removed.
//   oldEntry An object whose data field is null.
// Returns: The root node of the resulting tree; if anEntry matches
//          an entry in the tree, oldEntry's data field is the entry
//          that was removed from the tree; otherwise it is null.
private BinaryNode<T> removeEntry(BinaryNode<T> rootNode, T anEntry,
                                    ReturnObject oldEntry)
{
    if (rootNode != null)
    {
        T rootData = rootNode.getData();
        int comparison = anEntry.compareTo(rootData);
        if (comparison == 0)      // anEntry == root entry
        {
            oldEntry.set(rootData);
            rootNode = removeFromRoot(rootNode);
        }
        else if (comparison < 0) // anEntry < root entry
        {
            BinaryNode<T> leftChild = rootNode.getLeftChild();
            BinaryNode<T> subtreeRoot = removeEntry(leftChild, anEntry, oldEntry);
            rootNode.setLeftChild(subtreeRoot);
        }
        else                  // anEntry > root entry
        {
            BinaryNode<T> rightChild = rootNode.getRightChild();
            // A different way of coding than for left child
            rootNode.setRightChild(removeEntry(rightChild, anEntry, oldEntry));
        } // end if
    } // end if
    return rootNode;
} // end removeEntry

```

算法 `removeFromRoot`。前面的方法 `removeEntry` 通过调用方法 `removeFromRoot` 删除给定子树根中的项。在方法中，不论根结点有 0 个、1 个还是 2 个孩子，都根据段 26.20 到段 26.27 中的讨论来处理。如果给定的结点最多只有一个孩子，则可以直接删除结点和它的项。要删除有两个孩子的结点中的项，必须找到结点左子树中的最大项。删除含该最大项的结点。然后用最大项替代被删除的项。

下列算法概述了这些步骤。

```

Algorithm re moveFromRoot(rootNode)
// Removes the entry in a given root node of a subtree.
if (rootNode有两个孩子)
{
    largestNode = 在rootNode的左子树中有最大项的结点
}

```

26.30

26.31

```

    使用largestNode中的项替换rootNode中的项
    从树中删除largestNode
}
else if (rootNode有右孩子)
    rootNode = rootNode的右孩子
else
    rootNode = rootNode的左孩子 // Possibly null
// Assertion: If rootNode was a leaf, it is now null
return rootNode

```

- 26.32 私有方法 `removeFromRoot`。前面算法的实现中，调用了私有方法 `findLargest` 及 `removeLargest`，我们马上就来写相关的代码。虽然 `removeFromRoot` 不是递归的，但 `findLargest` 和 `removeLargest` 都是递归的。

给定子树的根，`removeFromRoot` 返回删除结点后那棵子树的根。

```

// Removes the entry in a given root node of a subtree.
// rootNode is the root node of the subtree.
// Returns the root node of the revised subtree.
private BinaryNode<T> removeFromRoot(BinaryNode<T> rootNode)
{
    // Case 1: rootNode has two children
    if (rootNode.hasLeftChild() && rootNode.hasRightChild())
    {
        // Find node with largest entry in left subtree
        BinaryNode<T> leftSubtreeRoot = rootNode.getLeftChild();
        BinaryNode<T> largestNode = findLargest(leftSubtreeRoot);
        // Replace entry in root
        rootNode.setData(largestNode.getData());
        // Remove node with largest entry in left subtree
        rootNode.setLeftChild(removeLargest(leftSubtreeRoot));
    } // end if

    // Case 2: rootNode has at most one child
    else if (rootNode.hasRightChild())
        rootNode = rootNode.getRightChild();
    else
        rootNode = rootNode.getLeftChild();
    // Assertion: If rootNode was a leaf, it is now null
    return rootNode;
} // end removeFromRoot

```

- 26.33 私有方法 `findLargest`。有最大项的结点将位于二叉查找树的最右结点。所以只要结点有右孩子，我们就去查找以那个孩子为根的子树。下面的递归方法对给定的树完成这个查找。

```

// Finds the node containing the largest entry in a given tree.
// rootNode is the root node of the tree.
// Returns the node containing the largest entry in the tree.
private BinaryNode<T> findLargest(BinaryNode<T> rootNode)
{
    if (rootNode.hasRightChild())
        rootNode = findLargest(rootNode.getRightChild());
    return rootNode;
} // end findLargest

```

- 26.34 私有方法 `removeLargest`。要删除有最大项的结点，不能简单地调用 `findLargest` 然后删除返回的那个结点。仅知道结点的引用是不能从树中删除它的。我们还必须知道其父结点的引用。下面的递归方法删除有最大项的结点——即最右结点——但不幸的是，它必须重复地进行查找，而这个工作 `findLargest` 才刚刚执行过。

```

// Removes the node containing the largest entry in a given tree.
// rootNode is the root node of the tree.
// Returns the root node of the revised tree.
private BinaryNode<T> removeLargest(BinaryNode<T> rootNode)
{
    if (rootNode.hasRightChild())
    {
        BinaryNode<T> rightChild = rootNode.getRightChild();
        rightChild = removeLargest(rightChild);
        rootNode.setRightChild(rightChild);
    }
    else
        rootNode = rootNode.getLeftChild();
    return rootNode;
} // end removeLargest

```

方法的开始很像是 `findLargest`。为了从给定树中删除最右结点，要从树的右子树中删除最右结点。递归调用会返回修改后的子树的根。这个根必须成为原树根的右孩子。

当树根没有右孩子时，返回左孩子，实际上删除了根。注意，这个递归方法并没有明确记下当前右孩子的父结点。而是将这个父结点的引用保存在递归的隐式栈中。

 **注：**前面这个从二叉查找树中删除一项的递归方法具有代表性。像 Java 这样的语言使用按值调用来传递参数，这使得这个递归实现复杂化，因为必须强制方法返回对根结点的引用。你可能会发现下面这个迭代方法更容易理解。注意到，它删除含有中序前驱的结点，但不需要重复地去查找它，故迭代版本的 `remove` 比递归版本的效率更高。

迭代实现

算法。回忆一下，传给方法 `remove` 的项，是与要从树中删除的项相匹配的项。所以 `remove` 的第一步是在树中进行查找。找到其数据与所给项相等的结点，如果那个结点有父结点，记下父结点。我们是删除找到的这个结点还是删除另一个结点，要依赖于它的孩子的个数。虽然段 26.19 列出了 3 种可能的情况，不过我们还是将它们归结为如下的两种情形：

- 1) 结点有两个孩子。
- 2) 结点最多有一个孩子。

第二种情形中，我们删除结点本身。但如果结点有两个孩子，则删除另一个最多有一个孩子的结点。换句话说，我们将第 1 种情形转化为第 2 种情形。

下列伪代码描述了 `remove` 要做的事情。

```

Algorithm remove(anEntry)
result = null
currentNode = 含有与anEntry值相等的值的结点
parentNode = currentNode的父结点
if (currentNode != null) // That is, if entry is found
{
    result = currentNode的数据 (要从树中删除的项)
    // Case 1
    if (currentNode有两个孩子)
    {
        // 得到要删除的结点及其父结点
        nodeToRemove = 含有anEntry的中序前驱的结点; 它最多有一个孩子
        parentNode = nodeToRemove的父结点
        将nodeToRemove中的项拷贝给currentNode
    }
}

```

26.35

```

    currentNode = nodeToRemove
    // Assertion: currentNode is the node to be removed; it has at most one child
    // Assertion: Case 1 has been transformed to Case 2
}
// Case 2: currentNode has at most one child
从树中删除currentNode
}
return result

```

26.36

公有方法 `remove`。我们将前一个算法中最重要的步骤实现为 `remove` 能够调用的私有方法。私有方法 `findNode` 找到含有与给定项相匹配的结点。因为需要指向那个结点及指向其父结点的引用，故 `findNode` 返回一对结点。为此，我们设计了一个私有类 `NodePair`，它含有构造方法和访问方法 `getFirst` 及 `getSecond`。`NodePair` 将是类 `BinarySearchTree` 的内部类。

私有方法 `getNodeToRemove` 找到含有给定结点中项的中序前驱的结点。因为我们还需要结点的父结点，所以方法返回作为类 `NodePair` 实例的一对结点。

最后，私有方法 `removeNode` 删除最多有一个孩子的结点。将结点及其父结点（如果存在）的引用传给方法。

使用这些私有方法，`remove` 的实现如下所示。

```

public T remove(T anEntry)
{
    T result = null;
    // Locate node (and its parent) that contains a match for anEntry
    NodePair pair = findNode(anEntry);
    BinaryNode<T> currentNode = pair.getFirst();
    BinaryNode<T> parentNode = pair.getSecond();
    if (currentNode != null)           // Entry is found
    {
        result = currentNode.getData(); // Get entry to be removed
        // Case 1: currentNode has two children
        if (currentNode.hasLeftChild() && currentNode.hasRightChild())
        {
            // Replace entry in currentNode with the entry in another node
            // that has at most one child; that node can be deleted
            // Get node to remove (contains inorder predecessor; has at
            // most one child) and its parent
            pair = getNodeToRemove(currentNode);
            BinaryNode<T> nodeToRemove = pair.getFirst();
            parentNode = pair.getSecond();
            // Copy entry from nodeToRemove to currentNode
            currentNode.setData(nodeToRemove.getData());
            currentNode = nodeToRemove;
            // Assertion: currentNode is the node to be removed; it has at
            // most one child
            // Assertion: Case 1 has been transformed to Case 2
        } // end if
        // Case 2: currentNode has at most one child; delete it
        removeNode(currentNode, parentNode);
    } // end if
    return result;
} // end remove

```

26.37

私有方法 `findNode`。要找到含有与给定项相等的结点，在循环中使用 `compareTo` 方法来比较给定项与树中的其他项。方法返回指向所要找的结点及其父结点的一对引用，这是

NodePair 类的实例。故 findNode 的格式如下所示。

```
private NodePair findNode(T anEntry)
{
    NodePair result = new NodePair();
    boolean found = false;
    ...
    if (found)
        result = new NodePair(currentNode, parentNode);
        // Located entry is currentNode.getData()
    return result;
} // end findNode
```

findNode 的实现细节留作练习。



学习问题 9 完成方法 findNode 的实现。

26.38

私有方法 getNodeToRemove。remove 找到含有要从树中删除项的结点后，根据结点的孩子个数进行处理。如果结点有两个孩子，则 remove 必须删除另一个最多有一个孩子的结点。私有方法 getNodeToRemove 找到这个结点。特别是，方法实现了段 26.25 中给出的伪代码的第一步。

找到 N 的左子树中的最右结点 R

其中，结点 N 是 currentNode，而结点 R 是 rightChild。

这一步的细节由下列伪代码描述。

```
// Find the inorder predecessor by searching the left subtree; it will be the largest
// entry in the subtree, occurring in the node as far right as possible
leftSubtreeRoot = currentNode 的左孩子
rightChild = leftSubtreeRoot
priorNode = currentNode
while (rightChild 有右孩子)
{
    priorNode = rightChild
    rightChild = rightChild 的右孩子
}
// Assertion: rightChild is the node to be removed and has no more than one child
```

下列 Java 代码实现了 getNodeToRemove。

```
private NodePair getNodeToRemove(BinaryNode<T> currentNode)
{
    // Find node with largest entry in left subtree by
    // moving as far right in the subtree as possible
    BinaryNode<T> leftSubtreeRoot = currentNode.getLeftChild();
    BinaryNode<T> rightChild = leftSubtreeRoot;
    BinaryNode<T> priorNode = currentNode;
    while (rightChild.hasRightChild())
    {
        priorNode = rightChild;
        rightChild = rightChild.getRightChild();
    } // end while
    // rightChild contains the inorder predecessor and is the node to
    // remove; priorNode is its parent
    return new NodePair(rightChild, priorNode);
} // end getNodeToRemove
```

26.39 私有方法 `removeNode`。最后一个方法假定，要删除的结点——称为 `nodeToRemove`——最多有一个孩子。如果 `nodeToRemove` 不是根，则 `parentNode` 是其父结点。

方法的开头，将 `childNode` 设置为 `nodeToRemove` 的孩子，如果存在。如果 `nodeToRemove` 是叶结点，则将 `childNode` 设置为 `null`。当结点是树根时，方法根据下列情形删除 `nodeToRemove`，如下所示。

```
if(nodeToRemove 是树根)
    将 childNode 作为树根
else
    将 parentNode 与 childNode 链接，所以删除了 nodeToRemove
```

如果将 `childNode` 作为树根，若 `nodeToRemove` 是叶结点，则意识到，我们将根正确地设置为 `null`。

`removeNode` 的实现如下。

```
private void removeNode(BinaryNode<T> nodeToRemove, BinaryNode<T> parentNode)
{
    BinaryNode<T> childNode;
    if (nodeToRemove.hasLeftChild())
        childNode = nodeToRemove.getLeftChild();
    else
        childNode = nodeToRemove.getRightChild();
    // Assertion: If nodeToRemove is a leaf, childNode is null
    if (nodeToRemove == get rootNode())
        setRootNode(childNode);
    else if (parentNode.getLeftChild() == nodeToRemove)
        parentNode.setLeftChild(childNode);
    else
        parentNode.setRightChild(childNode);
} // end removeNode
```

操作效率

26.40 操作 `add`、`remove` 和 `getEntry` 中的每一个都要从树根开始查找。当添加一个项时，如果项不在树中，则查找终止于叶结点；否则，查找可能提早结束。当删除或获取一项时，如果找不到，则查找终止于叶结点；成功查找可能提早结束。所以最坏情况下，这些查找从根开始，检查终止于叶结点的一条路径上的每个结点。从根到叶结点的最长路径的长度等于树的高度。所以每个操作所需的最多比较次数与树高 h 成正比。即操作 `add`、`remove` 和 `getEntry` 都是 $O(h)$ 的。

回忆一下，几棵不同的二叉查找树可能含有相同的数据。图 26-13 中含有两棵这样的树。图 26-13a 是使用这些数据能够创建的最低的二叉查找树；图 26-13b 是这样的树中最高的。

如果含 n 个结点，则最高的树的高度是 n 。事实上，这棵树看起来像是一个链表，查找它像是查找一个链表。后者是 $O(n)$ 的操作。所以这棵树上的 `add`、`remove` 和 `getEntry` 操作也是 $O(n)$ 的操作。

与之相反，最低的树是满树。查找这样的树是最有效的。在第 24 章看到，含有 n 个结点的满树的高度是 $\log_2(n+1)$ 。故最坏情况下，查找满二叉查找树是 $O(\log n)$ 操作。所以这种情况下，`add`、`remove` 和 `getEntry` 操作是 $O(\log n)$ 的。



学习问题 10 使用大 O 表示，方法 `contains` 的时间复杂度是多少？

学习问题 11 使用大 O 表示，方法 `isEmpty` 的时间复杂度是多少？

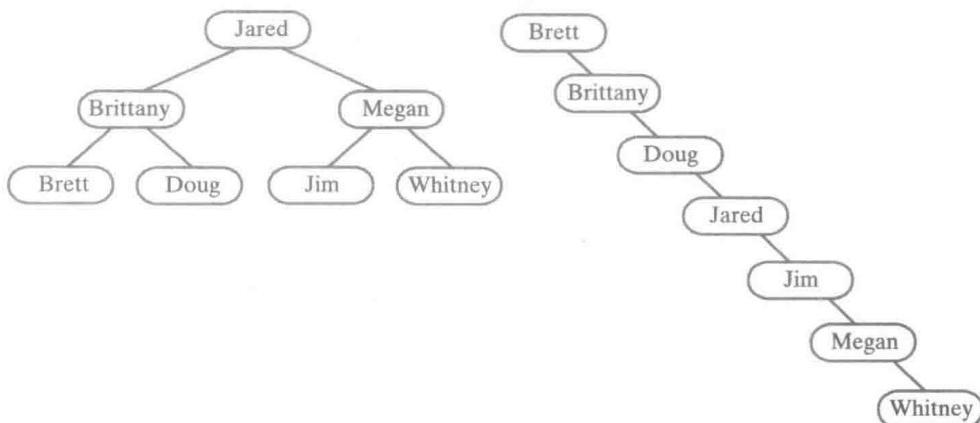


图 26-13 含有相同数据的两棵二叉查找树

平衡的重要性

不一定非得是满二叉查找树才具有添加、删除和获取操作的 $O(\log n)$ 性能。例如，如果我们从满树中删除一些叶结点，则这些操作的性能也不会改变。特别地，完全树也能有 $O(\log n)$ 的性能。26.41

恰巧的是，第 24 章段 24.10 所介绍的平衡概念，会影响一棵具体查找树的性能。事实上，如果二叉查找树是高度平衡的，则树上的添加、删除和获取操作会有 $O(\log n)$ 的性能。当创建一棵二叉查找树时，我们当然想让它高度平衡的。不幸的是，添加或删除项时可能会破坏二叉查找树的平衡性，因为这些操作影响了树形。

结点按什么次序添加

如果能正确回答段 26.12 中的学习问题 6，则你已经知道了向二叉查找树中添加项的次序会影响到树的形状。当向初始为空的树中添加项来创建二叉查找树时，知道这一点很重要。26.42

例如，假定由一组给定的数据创建图 26-13a 所示的满二叉查找树。通常这样的数据集是有序的，所以有理由假定名字是按字典序排列的。现在设想，我们定义了一棵空二叉查找树，然后将名字按如下的字典序添加进来：Brett、Brittany、Doug、Jared、Jim、Megan 和 Whitney。图 26-13b 所示为这些添加操作后得到的树。在能创建的树中，这是最高的树，且是操作效率最低的。



注：如果将项添加到初始为空的二叉查找树中，不要按有序的次序添加。

我们应该按什么次序来添加呢？Jared 是图 26-13a 所示树的根，所以先添加 Jared。接下来添加 Brittany，再然后是 Brett 和 Doug。最后添加 Megan、Jim 和 Whitney。显然，使用这个次序添加能得到图 26-13a 所示的树，但我们如何能提前决定呢？观察这些名字的字典序集合，注意到，Jared 恰在中间。我们先添加 Jared。Brittany 在左半部分数据集的中间，26.43

所以接下来添加 Brittany。由 Brittany 划分的每一半中都只含有一个名字，所以接下来添加它们。对排在 Jared 后面的名字——即数据集的右半部分重复这个过程。

我们不应该做这么多！实际上，如果我们按随机次序将数据添加到二叉查找树中，可以预料能有一棵操作能达到 $O(\log n)$ 的树。或许它不是所有能创建的树中最低的树，但它会接近于最低的。

二叉查找树的操作可以确保树仍是一棵二叉查找树。不幸的是，它们不能确保树保持平衡性。第 28 章介绍能保持平衡性从而也能保证效率的查找树。

ADT 字典的实现

26.44

可以使用本章迄今为止介绍的思想来实现 ADT 字典。回忆第 20 章的内容，字典中保存查找关键字及与关键字对应的值。例如，假定你想有一个名字与电话号码的字典。用 ADT 字典的术语来说，名字可以是查找关键字，而电话号码可能是相应的值。要获取一个电话号码，可以提供一个名字，字典将返回它的值。

下面是段 20.4 给出的用于字典的接口，但没有注释。

```
import java.util.Iterator;
public interface DictionaryInterface<K, V>
{
    public V add(K key, V value);
    public V remove(K key);
    public V getValue(K key);
    public boolean contains(K key);
    public Iterator<K> getKeyIterator();
    public Iterator<V> getValueIterator();
    public boolean isEmpty();
    public int getSize();
    public void clear();
} // end DictionaryInterface
```

第 21 章提出了 ADT 字典的几种实现方式。使用平衡查找树来存储字典项，字典的这种实现方式，是原方式的有吸引力的替代方案。作为这种实现方式的示例，我们将使用二叉查找树，虽然在添加或删除后它可能不再保持平衡性。第 28 章提出永远平衡的查找树，能用来替代实现字典。

26.45

数据项。我们需要一个含有查找关键字和对应值的数据对象的类。类 Entry——类似于第 21 章在基于数组实现 ADT 字典时使用的类——适合于我们的目的。这里让 Comparable 类定义方法 `compareTo`。这个方法通过比较查找关键字来比较 Entry 的两个实例。所以，字典的查找关键字必须属于 Comparable 类。

类 Entry 可以是类 `BstDictionary` 的私有的内部类，如程序清单 26-3 所示。该程序还列出了 `BstDictionary` 的数据域——一棵二叉查找树——及为树分配空间的构造方法。注意，Entry 是如何用在树的声明和分配中的。

程序清单 26-3 使用二叉查找树实现 ADT 字典的类框架

```
1 import TreePackage.SearchTreeInterface;
2 import TreePackage.BinarySearchTree;
3 import java.util.Iterator;
4 public class BstDictionary<K extends Comparable<? super K>, V>
5     implements DictionaryInterface<K, V>
6 {
7     private SearchTreeInterface<Entry<K, V>> bst;
```

```

8
9     public BstDictionary()
10    {
11        bst = new BinarySearchTree<>();
12    } // end default constructor
13
14    < Methods that implement dictionary operations are here. >
15    .
16
17    private class Entry<S extends Comparable<? super S>, T>
18        implements Comparable<Entry<S, T>>
19    {
20        private S key;
21        private T value;
22
23        private Entry(S searchKey, T dataValue)
24        {
25            key = searchKey;
26            value = dataValue;
27        } // end constructor
28
29        public int compareTo(Entry<S, T> other)
30        {
31            return key.compareTo(other.key);
32        } // end compareTo
33
34    < The class Entry also defines the methods equals, toString, getKey, getValue,
35    and setValue; no setKey method is provided. >
36    .
37    } // end Entry
38 } // end BstDictionary

```

BstDictionary 方法。方法 add 将给定的查找键和值封装为 Entry 的一个实例，并传给 BinarySearchTree 的 add 方法。然后使用这个方法返回的项来形成自己的返回值。BstDictionary 的 add 方法实现如下。

```

public V add(K key, V value)
{
    Entry<K, V> anEntry = new Entry<>(key, value);
    Entry<K, V> returnedEntry = bst.add(anEntry);

    V result = null;
    if (returnedEntry != null)
        result = returnedEntry.getValue();

    return result;
} // end add

```

remove 和 getValue 的实现类似于 add 的实现。因为这些方法都只有一个查找键作为参数，故它们生成的 Entry 实例将查找键和 null 值封装在一起。例如，remove 最初的几行代码如下：

```

public V remove(K key)
{
    Entry<K, V> findEntry = new Entry<>(key, null);
    Entry<K, V> returnedEntry = bst.remove(findEntry);

```

结尾的代码类似于 add 方法。方法 getValue 的实现与 remove 是一样的，除了在 BinarySearchTree 中调用 getEntry 而不是调用 remove 之外。

通过调用 BinarySearchTree 中相应的方法，可以实现 getSize、isEmpty、contains 和 clear 等方法。将这些实现留作练习。



学习问题 12 通过调用 `BinarySearchTree` 中的相应方法，实现 `BstDictionary` 中的每个方法 `getSize`、`isEmpty`、`contains` 和 `clear`。

学习问题 13 通过调用 `BstDictionary` 的方法 `getValue`，给出方法 `contains` 的另一种实现。

26.47

迭代器。 `DictionaryInterface` 中规范说明了返回迭代器的两个方法。方法 `getKeyIterator` 返回的迭代器，可按有序次序访问查找键；`getValueIterator` 返回的迭代器，能提供属于这些查找键的值。例如，`getKeyIterator` 的实现如下。

```
public Iterator<K> getKeyIterator()
{
    return new KeyIterator();
} // end getKeyIterator
```

类 `KeyIterator` 是 `BstDictionary` 的内部类，它用到了 `BinarySearchTree` 中的方法 `getInorderIterator`。其实现如下。

```
private class KeyIterator implements Iterator<K>
{
    Iterator<Entry<K, V>> localIterator;
    public KeyIterator()
    {
        localIterator = bst.getInorderIterator();
    } // end default constructor
    public boolean hasNext()
    {
        return localIterator.hasNext();
    } // end hasNext
    public K next()
    {
        Entry<K, V> nextEntry = localIterator.next();
        return nextEntry.getKey();
    } // end next
    public void remove()
    {
        throw new UnsupportedOperationException();
    } // end remove
} // end KeyIterator
```

你可以用类似的方式实现 `getValueIterator`。

26.48

注释。 ADT 字典的这个实现，与作为底层的查找树有同样的时间效率。当二叉查找树是平衡的时，操作是 $O(\log n)$ 的。但当添加或删除项时，二叉查找树可能会失去它的平衡性，这样，字典操作的效率会降为 $O(n)$ 。在第 28 章将会看到的保持平衡的查找树，将能更好地实现字典，优于目前所给的实现。

还要注意到，二叉查找树按查找键有序来保存字典项。因此，`getKeyIterator` 能按序遍历查找键。相反，其他的字典实现——例如散列——无序遍历查找关键字。

本章小结

- 二叉查找树是一棵二叉树，其结点含有 `Comparable` 类型的对象。对树中的每个结点，
 - ◆ 结点中的数据大于结点左子树中的数据。
 - ◆ 结点中的数据小于（或等于）结点右子树中的数据。

- 查找树除了具有所有树共有的操作外，还有 `contains`、`getEntry`、`add`、`remove` 和 `getInorderIterator` 等操作。
- 类 `BinarySearchTree` 可以是 `BinaryTree` 的子类，但必须禁用 `setTree`。为避免改变树中结点的次序，客户必须只能使用 `add` 方法来创建二叉查找树。
- 在二叉查找树中查找项的查找算法，构成了方法 `getEntry`、`add` 和 `remove` 的基础。这些方法中的每一个都有迭代和递归的实现方式。
- 每次将项添加到二叉查找树中，就是在树中添加了一个叶结点。新项位于查找算法能找到它的地方。
- 从二叉查找树中删除一个项，要根据项所在的那个结点拥有的孩子个数来处理。当结点是叶结点或有一个孩子时，可以删除结点本身。当结点有孩子存在时，结点的父结点将这个唯一的孩子作为自己的孩子。但当结点 N 有两个孩子时，要用另一个容易删除的结点 r 的项替换结点中的项。为了维护二叉查找树中的次序，项 r 可以是 N 左子树中最大的项，也可以是 N 右子树中最小的项。而 r 所在的结点或者是叶结点，或者是有一个孩子的结点。
- 在二叉查找树上的获取、添加和删除操作，最快为 $O(\log n)$ 的，最慢为 $O(n)$ 的。查找的性能依赖于树的形状。当树是高度平衡树时，二叉查找树上的操作是 $O(\log n)$ 的。
- 将项添加到二叉查找树中的次序影响了树的形状，所以也影响了它的平衡性。随机添加，相对于按序添加来说，更能得到一棵平衡的树。
- 使用二叉查找树能实现 ADT 字典。虽然它的实现不难写，但如果添加和删除破坏了树的平衡性，其效率会变差。

练习

1. 将下列查找键添加到初始为空的二叉查找树中：10、5、6、13、15、8、14、7、12、4，给出得到的结果。
2. 将查找键 10、5、6、13、15、8、14、7、12、4 添加到初始为空的二叉查找树中，以什么样的次序添加能得到最平衡的树？
3. 将查找键 10、5、6、13、15、8、14、7、12、4 添加到初始为空的二叉查找树中，给出 4 种能得到最不平衡树的不同的添加次序。
4. 第 14 章的图 14-4a 给出了 Fibonacci 数列 F_n 的递归计算过程。这棵树是高度平衡树吗？
5. 实现迭代的 `getEntry` 方法。
6. 从图 26-11a 所示的二叉查找树中删除 Doug。然后再以两种不同的方法删除 Chad。
7. 以两种不同的方法从图 26-11d 所示的二叉查找树中删除 Doug。
8. 假定有两个孩子的结点含有项 `target`，如图 26-9a 所示。说明，如果用其中序后继 `succ` 来替代 `target`，然后再删除含有 `succ` 的结点，仍将得到一棵二叉查找树。
9. 为什么二叉查找树的中序遍历能以查找键有序的次序访问结点？使用段 26.1 给出的二叉查找树的定义来解释。
10. 考虑图 26-13a 所示的满二叉查找树。假定遍历树并将结果数据保存在一个文件中。如果随后读入该文件并将数据添加到初始为空的二叉查找树中，如果刚才的遍历是下面的 4 种遍历，得到的树分别是什么？
 - a. 前序
 - b. 中序
 - c. 层序
 - d. 后序
11. 假定遍历一棵二叉查找树并将它的数据保存在一个文件中。如果随后读入该文件并将数据添加到

初始为空的二叉查找树中，用什么样的遍历写文件时，能得到如下的新树？

- a. 尽可能高
- b. 与原二叉查找树一样的树

12. 段 26.17 中给出了方法 `addEntry` 的迭代实现算法。用下列算法实现该方法。

```
Algorithm addEntry(binarySearchTree, anEntry)
result = null
currentNode = binarySearchTree 的根结点
parentNode = null
while (未找到anEntry且currentNode不是null)
{
    if (anEntry等于currentNode中的项)
    {
        result = currentNode中的项
        将currentNode中的项替换为anEntry
    }
    else if (anEntry < currentNode中的项)
    {
        parentNode = currentNode
        currentNode = currentNode 的左孩子
    }
    else // anEntry > entry in currentNode
    {
        parentNode = currentNode
        currentNode = currentNode 的右孩子
    }
}
if (在树中未找到anEntry)
{
    创建一个新结点，将anEntry放到结点中
    if (anEntry < parentNode中的项)
        令新结点是parentNode的左孩子
    else
        令新结点是parentNode的右孩子
}
return result
```

13. 段 26.28 到 段 26.34 中 描 述 的 方 法 `remove` 和 递 归 的 `removeEntry`，用 到 了 内 部 类 `ReturnObject`。采 用 这 种 方 式，`removeEntry` 可 以 将 改 变 后 的 树 根 和 删 除 的 项 都 传 递 给 `remove`。使 用 Java 插 曲 8 中 那 样 的 类 `Pair<S, T>` 修改 这 些 方 法。`Pair` 类 需 要 对 其 数据 域 的 访 问 方 法。这 样，方 法 `removeEntry` 可 以 将 根 和 被 删 除 的 项 组 成 `Pair` 对 象 而 返回。
14. 段 26.43 由 一 组 具 体 的 查 找 键 创建 了一 棵 平 衡 的 二 叉 查 找 树。总 结 这 个 方 法，编 写 由 含 n 个 元 素 的 有 序 集 合 创建 平 衡 二 叉 查 找 树 的 递 归 方 法。
15. 编 写 回 而 二 叉 查 找 树 中 最 小 查 找 键 的 算 法。
16. 从 段 26.23 开 始，你 了 解 了 如 何 找 到 有 两 个 孩 子 结 点 的 中 序 前 驱 或 是 中 序 后 继。不 幸 的 是，这 个 方 法 不 能 用 于 叶 结 点。对 于 有 一 个 孩 子 的 结 点，这 个 办 法 能 找 到 其 前 驱 或 是 后 继，但 不 能 找 到 两 个。讨 论 如 何 改 变 结 点 的 结 构，使 之 可 以 找 到 任 意 结 点 的 中 序 前 驱 或 是 中 序 后 继。
17. 编 写 回 而 二 叉 查 找 树 中 第 二 大 值 的 算 法。
18. 为 什 么 用 二 叉 查 找 树 实 现 优 先 队 列 时 效 果 不 佳？
19. 考 虑 判 定 二 叉 查 找 树 是 否 是 第 24 章 段 24.10 描 述 的 高 度 平 衡 树 的 方 法。方 法 头 可 能 如 下：

```
public boolean isBalanced()
```

为 类 `BinarySearchTree` 编 写 这 个 方 法。它 应 该 调 用 同 名 的 私 有 递 归 方 法。

20. 编 写 一 个 静 态 方 法，接 受 一 个 `BinaryTree` 对 象 的 参 数，如 果 参 数 表 示 的 树 是 二 叉 查 找 树，则 方 法 返回 `true`。对 给 定 的 树 中 的 每 个 结 点 仅 检 查 一 次。

21. 考虑对相同的查找键允许有重复项的两棵空二叉查找树。对其中的一棵树，添加 m 个互异的项，每个项都有一个不同的查找键。对另一棵树，对 m 个项中的每一个都添加 k 次，总共有 $m \times k$ 个项。假定每个项保存在一个结点中，比较两棵树的高度。讨论第二棵树中项的添加次序如何影响它的高度。给出能得到最高树及最低树的添加次序。
22. 段 26.4 描述了允许有重复值的二叉查找树。将项的重复值放在项的右子树中。
 - a. 这种机制的优缺点分别是什么？
 - b. 假定修改二叉查找树的定义，允许项的重复值在项的右子树中或左子树中。如果随机选择子树，这种机制的优缺点分别是什么？

项目

1. 规范说明并实现允许重复值的二叉查找树的类。如段 26.4 中提出的，将项的重复值放在项的右子树中。提供一个方法，对给定的项在树中进行查找并返回首次找到的项。还要提供一个类似的方法，返回与给定项相匹配的所有项的线性表。
2. 重做前一个项目，但允许将重复值随机放在项的左子树或右子树中。所以，修改二叉查找树的定义如下：
对二叉查找树中的每个结点，
 - 结点中的数据大于或等于结点左子树中的数据。
 - 结点中的数据小于或等于结点右子树中的数据。重复值的查找必须要在两棵子树中进行。
3. 使用二叉查找树实现 ADT 有序表。
4. 设计使用二叉查找树对对象数组进行排序的算法。这样的排序称为树排序 (tree sort)。实现并测试你的算法。讨论你的树排序的平均及最差时间复杂度。
5. 实现二叉查找树，包括练习 15 和练习 16 中提出的下列方法：

```

/** @return The entry with the smallest search key. */
public T getMin();

/** @return The entry with the largest search key. */
public T getMax();

/** @return Either the inorder predecessor of anEntry, or
     anEntry if it's the smallest item in the tree, or
     null if anEntry is not in the tree. */
public T getPredecessor(T anEntry);

/** @return Either the inorder successor of anEntry, or
     anEntry if it's the largest item in the tree, or
     null if anEntry is not in the tree. */
public T getSuccessor(T anEntry);

```

6. 实现派生于第 25 章项目 7 描述的 `ArrayBinaryTree` 的类 `ArrayBinarySearchTree`。
7. 编写 Java 代码，由 n 个随机整数创建一棵二叉查找树并返回查找树的高度。当 $n = 2^h - 1$ 时执行代码，其中 h 取值为 $4 \sim 12$ 。对随机创建的查找树的高度 h ，与最低的二叉查找树的高度进行比较。
8. 第 1 章将集合定义为不允许有重复值的包。使用二叉查找树来保存集合项，定义集合类。
9. 重做第 20 章项目 9，使用二叉查找树来实现两个字典。编写 Java 代码，在存储 Java 保留字的第一个字典里创建一棵平衡的二叉查找树。为什么包含 Java 保留字的查找树是平衡树是很重要的？你能保证用户自定义的标识符的查找树也是平衡的吗？
10. 重做第 25 章项目 9，使用二叉查找树替代 26 叉树。
11. 比较两棵二叉查找树当添加更多对象时的性能。初始时，一棵树是平衡的，而另一棵树不是平衡的。

首先修改 `BinarySearchTreeInterface` 和 `BinarySearchTree`，让 `add` 方法返回进行比较的次数。然后使用新版本的 `BinarySearchTree` 编写程序，具体步骤如下。创建两棵空

的二叉查找树。为每棵树分配两个变量。一个变量保存将值添加到树中时的比较次数之和，另一个变量保存在插入若干个值后的那一时刻树的高度之和。这些变量的名字分别是 `comparisonSum1`、`comparisonSum2`、`heightSum1` 和 `heightSum2`。

在执行 100 次的循环中，做下列事情：

- 将值 1000, 2000, 3000, 4000, 5000, 6000 和 7000 分别添加到两棵树中。在第一棵树中，按升序添加。在第二棵树中，按能得到完全树的次序添加。第一棵树不是平衡的，而第二棵树将是平衡的。
- 生成 10 个 0 ~ 8000 之间的随机数。将它们以相同的次序添加到每棵树中。每次添加后，使用每次添加时进行的比较次数更新每棵树的 `comparisonSum` 值。
- 将每棵树的高度加到 `heightSum` 变量中。
- 清除两棵树。

循环结束后，计算将值插入每棵树的平均比较次数。（对每棵树，用 `comparisonSum` 除以 1000。注 1000 等于迭代次数 100 乘以每次迭代时插入的值的个数 10。）还要计算每棵树每次插入后的平均高度。（每个变量 `heightSum` 除以 100。）显示并记录结果。

第二次执行程序，这次在每个循环迭代中添加 100 个 0 ~ 8000 之间的随机数。第三次执行程序，换成添加 1000 个随机数。讨论结果并得出结论。

12. **kd 树 (kd-tree)** 或 **k 维树 (k-dimensional tree)** 是一棵组织 k 维空间中点的二叉树。每个结点含有并表示一个 k 维点。每个不是叶子的结点 N ，对应于一个将空间分为两个部分的超平面[⊖]。超平面左边的点在 N 的左子树中，超平面右边的点在 N 的右子树中。基于 k 维空间和 **kd 树** 之间的关系，可使用树找到给定范围内的所有点——范围查找——或找到距离给定点最近的点——最近邻查找。

本项目中，选择 k 是 2，考虑 2 维空间及 2d 树，树中的结点含有该空间中的点。为避免术语“2d 树”可能引起的任何混乱，计算机科学家通常将树描述为“2 维 kd 树”。不过，这里我们使用更短的名字“2d 树”。

2d 树推广了二叉查找树，它根据数据点的 x 或 y 坐标来查找每个结点。根据结点插入树中的层来决定选择它的哪个坐标。插入空树中的第一个点，放到成为树根的结点中。如果要插入的下一个点，其 x 坐标小于根中点的 x 坐标，则将新点放到根的左孩子中。否则，将它放到根的右孩子中。下一层——层 3——的插入比较 y 坐标；层 4 的插入比较 x 坐标，以此类推。

例如，现在将点 (50, 40)、(40, 70)、(80, 20)、(90, 10) 和 (60, 30) 插入到初始为空的 2d 树中。图 26-14 显示了这棵树的构造过程。(a) 显示含有第一个点 (50, 40) 的根。(暂且不画树根下面的内容) 为将 (40, 70) 插入根的孩子中，比较点的 x 坐标 40，与根中点的 x 坐标 50。因为 40 小于 50，故新点放到根的左孩子中，如图 26-14b 所示。类似地，因为 80 大于 50，故将下一个点 (80, 20) 放到根的右孩子中 (图 26-14c)。要插入 (90, 10)，从树根开始，比较 x 坐标。因为 90 大于 50，故移动到根的右孩子，并比较 y 坐标。发现 10 小于 20，故 (90, 10) 放到根右孩子的左孩子中，如图 26-14d 所示。最后一个点 (60, 30)，使用类似的步骤插入，得到的树如图 26-14e 所示。

2d 树的图形含义画在了图 26-14 中各棵树的下面。从一个含有树中所有点的正方形开始。例如，如图 26-14a 所示的一个 100×100 的正方形，含有示例中的 5 个点。过树根中点的 x 坐标的一条垂直线，将正方形划分为两个区域。根的左子树中的任何点，将位于这条线的左边，而根的右子树中的点将位于这条线的右边。图 26-14b 显示过点 (40, 70) 的一条水平线。含有 (40, 70) 的结点的左子树中的点，位于这条水平线的上方及垂直线的左方；即它们位于原正方形的左上角

[⊖] k 维空间的超平面是一个由 k 个变量的单一线性方程表示的 $(k-1)$ 维面，它将空间分成两个区域。例如，2 维空间中，由变量 x 和 y 的线性方程所描述的一条直线将空间划分。3 维空间中，由变量 x 、 y 和 z 的线性方程所描述的一个平面将空间划分。

矩形内。

实现一棵 2d 树，至少提供一个将新点插入的方法，及一个检测给定点是否在树中的方法。

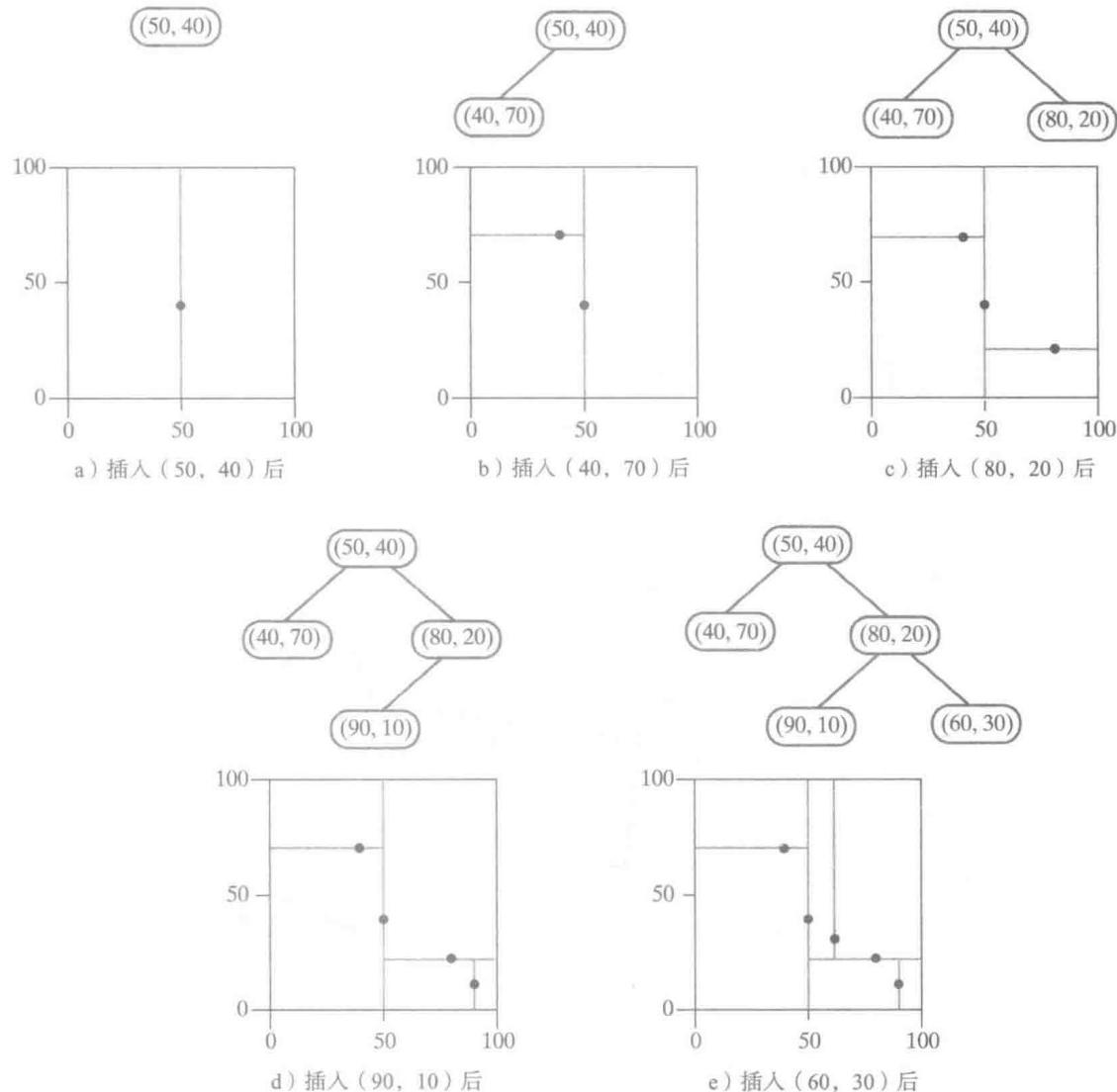


图 26-14 对 5 个给定点创建一棵 2d 树的步骤

13. 重做第 18 章项目 9，但使用二叉查找树替代有序表进行研究。考虑二叉查找树的方法 `add`、`remove`、`getEntry` 和 `contains`。比较类 `BinarySearchTree` 与 `LinkedList` 在第 18 章中两种不同实现版本的性能。
14. 重做前一个项目，但使用第 18 章项目 10 给出的混合操作。
15. 重做第 23 章项目 10，但采用变型后的拉链法进行实验。例如，散列表可能指向二叉查找树而不是一个链表。

堆的实现

先修章节：第 2 章、第 11 章、第 24 章

目标

学习完本章后，应该能够

- 使用数组表示堆
- 向基于数组的堆中添加一项
- 删除基于数组的堆的根
- 由给定的项创建堆
- 使用堆排序对数组进行排序

回忆第 24 章，堆是一棵其结点有特定排列次序的完全二叉树。当二叉树是完全树时，可以使用数组来高效简洁地表示它。堆最常见的实现是使用一个数组，这是本章要讨论的问题之一。

正如在第 24 章所见，使用堆能高效地实现 ADT 优先队列。在本章的后面，你会学习如何使用堆来排序数组。

再论：ADT 堆

27.1

堆是一棵完全二叉树，其结点含有 Comparable 对象。最大堆中，每个结点中的对象大于等于结点后代中的对象。段 24.33 给出了用于最大堆的接口，如下所示。

```
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
    public void add(T newEntry);
    public T removeMax();
    public T getMax();
    public boolean isEmpty();
    public int getSize();
    public void clear();
} // end MaxHeapInterface
```

我们在实现最大堆时会用到这个接口。



注：你或许听过“堆”这个词，它用来指执行 new 运算符时可分配给程序使用的内存单元集合。但那个堆不是我们将在本章讨论的 ADT 堆的实例。不过，在程序设计语言的书中会涉及那个概念。

使用数组表示堆

27.2

表示一棵完全二叉树。我们从使用数组来表示一棵完全二叉树开始。完全树是直到倒数第二层都是满的，且最后一层的叶结点从左至右填充。所以，到最后的叶结点之前，完全树没有空位。

假定我们依层序遍历访问结点的次序，对完全二叉树中的结点从1开始编号。图27-1a显示了使用这种方式编号的一棵树。现在假定我们将树的层序遍历结果放入数组从下标1开始的连续位置中，如图27-1b所示。树中数据的这种表示方式能让我们实现所需的树操作。从下标1开始，而不是从0开始，可以简化树的实现，后面会看到这一点。

因为树是完全树，所以通过简单地计算结点的编号，就可以找到任何结点的孩子结点或父结点。这个编号与结点对应的数组下标相同。所以结点*i*的孩子——如果存在——保存在数组下标 $2i$ 和 $2i+1$ 处。该结点的父结点在数组下标 $i/2$ 处，当然除非这个结点是根。那种情况下， $i/2$ 是0，因为根在下标为1处。要找到根，可以查看这个下标，或是一个特殊值——称为哨兵（sentinel）——它放在下标0处。

27.3

注：当二叉树是完全树时，使用数组而不是结点链表是令人满意的。可以依层序遍历将树的数据存储到数组连续的位置中。这样的表示法能让你快速找到结点的父结点或孩子结点。如果从数组下标为1处开始存储树——即如果你跳过数组的第一个元素——则对于数组下标*i*处的结点

- 其父结点在下标*i/2*处，除非结点是根（*i*是1）。
- 如果存在，孩子结点在下标 $2i$ 和 $2i+1$ 处。

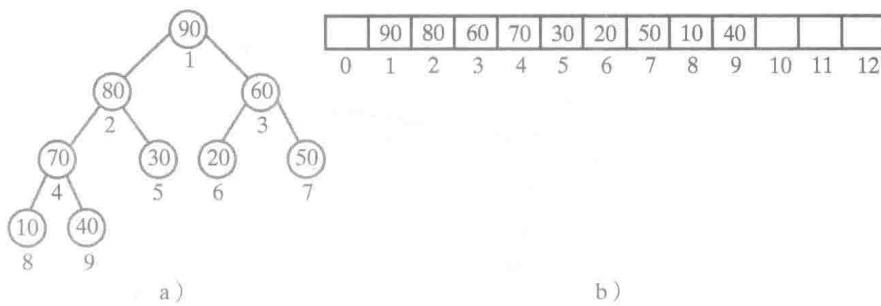


图 27-1 按层序次序对结点进行编号且使用数组表示的一棵完全二叉树

图27-1a中的完全二叉树实际上是一个最大堆。在最大堆的实现中，我们使用它的数组表示——如图27-1b所示。



学习问题 1 如果堆中的项按层序保存在数组从下标0开始的单元中，则哪个数组项表示结点的父结点、左孩子和右孩子？

27.4

开始实现 MaxHeap。如程序清单27-1所示，类的开头是下列数据域：堆中 Comparable 项所在的数组、数组中最后一项的下标及表示堆默认容量的一个常量。如果 lastIndex 小于1，则堆为空，因为堆从下标为1处开始。两个构造方法类似于前面见过的基于数组实现的类中的构造方法。我们多分配一个数组位置，因为第一个位置不使用。方法 getMax、isEmpty、getSize 和 clear 的实现很简单，见程序。下面考虑 add 和 removeMax 方法。

程序清单 27-1 部分完成的类 MaxHeap

```

1 import java.util.Arrays;
2 public final class MaxHeap<T extends Comparable<? super T>>
3     implements MaxHeapInterface<T>
4 {
5     private T[] heap;           // Array of heap entries

```

```
6     private int lastIndex; // Index of last entry
7     private boolean integrityOK = false;
8     private static final int DEFAULT_CAPACITY = 25;
9     private static final int MAX_CAPACITY = 10000;
10
11    public MaxHeap()
12    {
13        this(DEFAULT_CAPACITY); // Call next constructor
14    } // end default constructor
15
16    public MaxHeap(int initialCapacity)
17    {
18        // Is initialCapacity too small?
19        if (initialCapacity < DEFAULT_CAPACITY)
20            initialCapacity = DEFAULT_CAPACITY;
21        else // Is initialCapacity too big?
22            checkCapacity(initialCapacity);
23
24        // The cast is safe because the new array contains all null entries
25        @SuppressWarnings("unchecked")
26        T[] tempHeap = (T[]) new Comparable[initialCapacity + 1];
27        heap = tempHeap;
28        lastIndex = 0;
29        integrityOK = true;
30    } // end constructor
31
32    public void add(T newEntry)
33    {
34        <See Segment 27.8.>
35    } // end add
36
37    public T removeMax()
38    {
39        <See Segment 27.12.>
40    } // end removeMax
41
42    public T getMax()
43    {
44        checkIntegrity();
45        T root = null;
46        if (!isEmpty())
47            root = heap[1];
48        return root;
49    } // end getMax
50
51    public boolean isEmpty()
52    {
53        return lastIndex < 1;
54    } // end isEmpty
55
56    public int getSize()
57    {
58        return lastIndex;
59    } // end getSize
60
61    public void clear()
62    {
63        checkIntegrity();
64        while (lastIndex > -1)
65        {
66            heap[lastIndex] = null;
67            lastIndex--;
68        } // end while
69        lastIndex = 0;
70    } // end clear
```

```

71     < Private methods >
72     ...
73 } // end MaxHeap

```

添加项

基本算法。向堆中添加一项的算法并不难。回忆最大堆中，结点中的对象大于等于其后代对象。假定我们想向图 27-1 所示的最大堆中添加 85。首先应将新项作为树的下一个叶结点。图 27-2a 显示将 85 添加为 30 的左孩子。注意到，实际上是将 85 放在了图 27-1b 所示数组下标为 10 的位置。

图 27-2a 不再是一个堆，因为 85 不在应在的位置。要将树转换为堆，要让 85 上浮 (float up) 到其正确位置。因为 85 大于其父结点 30，故将它与父结点交换，如图 27-2b 所示。85 仍大于其新的父结点 80，所以再次交换 (图 27-2c)。现在 85 小于其父结点，所以已将图 27-2a 中的树转换为最大堆。

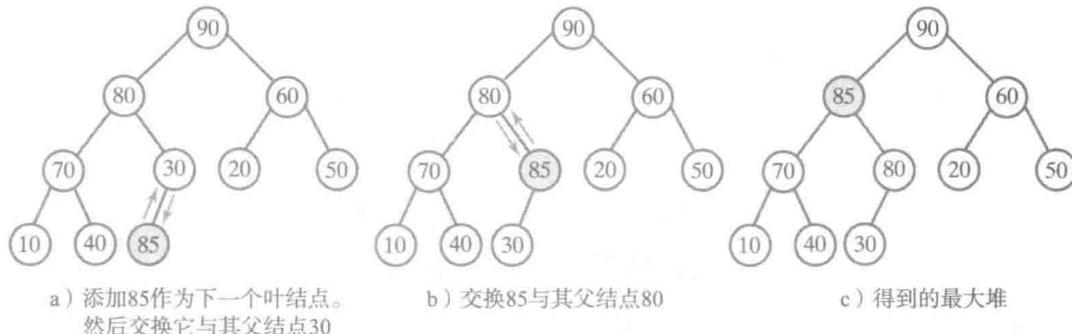


图 27-2 将 85 添加到图 27-1a 所示的最大堆中的步骤



学习问题 2 将 100 添加到图 27-2c 所示的堆中，需要哪些步骤？

避免交换。虽然在上面提到的交换使得算法更易理解也更方便描述，但实际上并不需要做这么多。我们不再将新项放在树中下一个可用的位置，如图 27-2a 中所做的那样，我们只需为其保留位置即可。在基于数组的实现中，只需检查数组不满。图 27-3a 中将新孩子表示为一个空圆圈。

然后将新项——本例中是 85——与新孩子的父结点相比较。因为 85 大于 30，故将 30 移到新孩子处，如图 27-3b 所示。将原来保存 30 的结点看作空的。现在比较 85 与空结点的父结点 80。因为 85 大于 80，故将 80 移到空结点中，如图 27-3c 所示。因为 85 不再大于下一个父结点 90，故将新项放到空结点中，如图 27-3d 所示。



注：要向堆中添加一项，从下一个可用于叶结点的位置开始。沿着从这个叶结点向根的路径进行处理，直到为新项找到正确位置为止。将项从父结点移到孩子结点处，如前面的操作那样，最终为新项找到空间。

图 27-4 从表示堆的数组的角度，展示了这些相同的步骤。图 27-4a 类似于图 27-3a，为新项在下标 10 处标注出空间。这个位置的父结点在位置 10/2 即 5 处。所以将新项 85 与下

标 5 位置的内容 30 进行比较。因为 $85 > 30$ ，所以将 30 移到下标 10 处（图 27-4b 和图 27-3b）。其余的步骤类似。注意，图 27-4d 对应于图 27-3c，而图 27-4f 对应于图 27-3d。

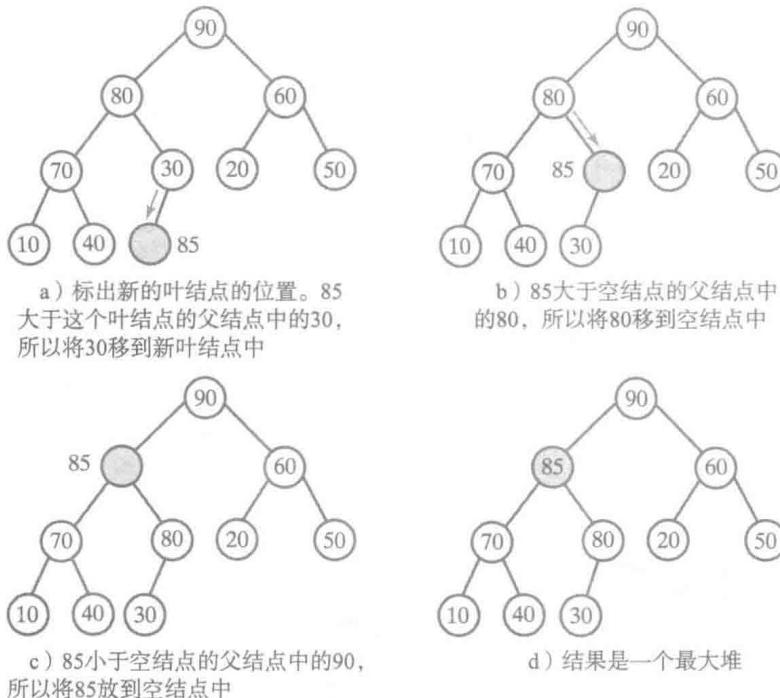


图 27-3 修改图 27-2 所示的添加 85 的步骤，以避免交换

27.7 改进算法。下列算法简述了将新项添加到堆中的步骤。为了忽略数组的第一个位置，我们只需确保 `parentIndex` 大于 0 即可。注意，添加后数组的大小按需扩大，如第 11 章 `AList` 的方法 `add` 中所做的一样。

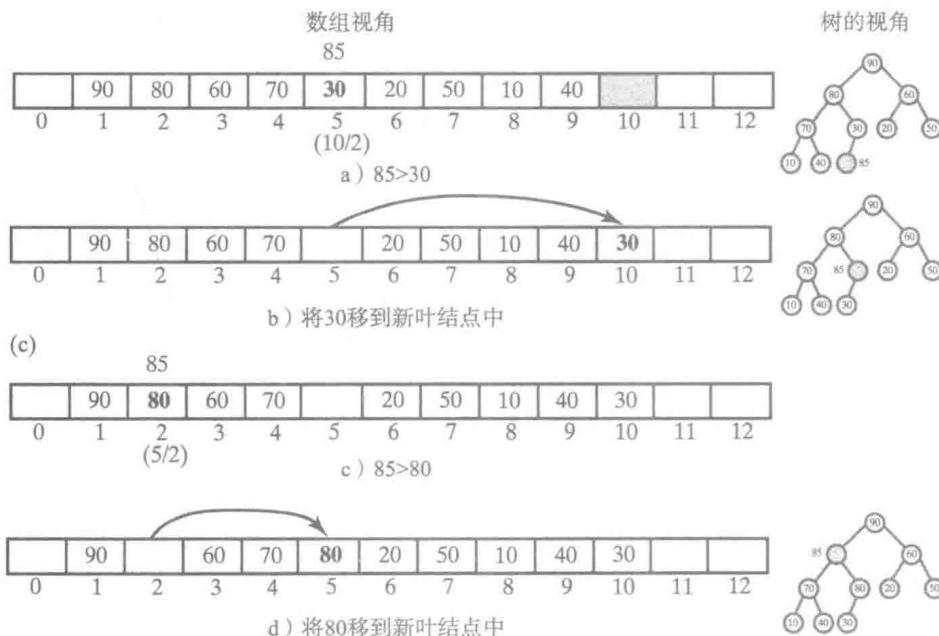


图 27-4 图 27-3 所示步骤的数组表示

| | | | | | | | | | | | | |
|---|------------|---|----|----|----|----|----|----|----|----|----|----|
| | 90 | | 60 | 70 | 80 | 20 | 50 | 10 | 40 | 30 | | |
| 0 | 1 (2/2) | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

e) $85 < 90$

| | | | | | | | | | | | | |
|---|----|----|----|----|----|----|----|----|----|----|----|----|
| | 90 | 85 | 60 | 70 | 80 | 20 | 50 | 10 | 40 | 30 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

f) 将 85 插入空位中

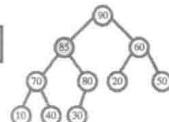


图 27-4 (续)

Algorithm add(newEntry)

// 前置条件：数组heap有空间放置另一个项。

```

newIndex = 下一个可用的数组位置的下标
parentIndex = newIndex/2           // Index of parent of available location
while (parentIndex > 0 且 newEntry > heap[parentIndex])
{
    heap[newIndex] = heap[parentIndex] // Move parent to available location

    // Update indices
    newIndex = parentIndex
    parentIndex = newIndex/2
}

heap[newIndex] = newEntry           // Place new entry in correct location
if (数组heap已满)
    倍增数组大小
  
```



学习问题 3 使用前一个不进行交换的算法重做学习问题 2。用树及数组显示堆的每一步。

方法 add。现在严格遵循前一个算法来实现方法 add。

27.8

```

public void add(T newEntry)
{
    checkIntegrity();          // Ensure initialization of data fields
    int newIndex = lastIndex + 1;
    int parentIndex = newIndex / 2;
    while ( (parentIndex > 0) && newEntry.compareTo(heap[parentIndex]) > 0)
    {
        heap[newIndex] = heap[parentIndex];
        newIndex = parentIndex;
        parentIndex = newIndex / 2;
    } // end while
    heap[newIndex] = newEntry;
    lastIndex++;
    ensureCapacity();
} // end add
  
```

如果将一个哨兵值放在数组未用的下标 0 处，就可以省略 while 语句中关于 parentIndex 的测试。可以用 newEntry 当哨兵值。应该回答学习问题 5，并能明白这个修改是正确的。

最差情况下，该方法沿从叶到根的路径执行。在第 24 章段 24.11 看到，有 n 个结点的完全树的高度是 $\log_2(n+1)$ 向上取整。所以最坏情况下，add 方法是 $O(\log n)$ 的。



学习问题4 定义私有方法 ensureCapacity。

学习问题5 修改前一个方法 add，将 newEntry 作为哨兵值放在数组未用的下标 0 处。然后可以省略 while 语句中关于 parentIndex 的测试。

删除根

27.9

基本算法。用于最大堆的 removeMax 方法删除并返回堆中最大的对象。这个对象是最大堆的根。我们来删除图 27-3d 所示堆根中的项。图 27-5a 显示了这个堆，好像它的根是空的。

我们不想将根结点从堆中去掉，因为这会留下两棵不相交的子树。相反我们删除叶结点，即堆中最后一个结点。为此，将叶结点中的数据 30 拷贝到根中，然后从树中删除叶结点，如图 27-5b 所示。当然，在基于数组的实现中，删除这个叶结点仅意味着调整 lastIndex 的值。

30 不在正确位置，所以得到的不再是堆。让 30 下沉 (sink down) 到其正确位置。只要 30 小于其孩子结点，就将它与其较大的孩子相交换。所以，在图 27-5c 中交换了 30 与 85。继续，交换 30 和 80，如图 27-5d 所示。本例中，30 定位于叶结点。一般的，不在正确位置的项将定位到其孩子不大于该项的结点处。



学习问题6 从图 27-5d 所示的堆中删除根的步骤是什么？

27.10

将半堆转换为堆。图 27-5b 中的树称为半堆 (semiheap)。除了根以外，半堆中的对象与它们在堆中的次序是一样的。在删除堆根的过程中，我们得到一个半堆，然后将它转换回堆。与方法 add一样，可以不交换项以节省时间开销。图 27-6 显示图 27-5b 所示的半堆及不进行交换将它转为堆的步骤。

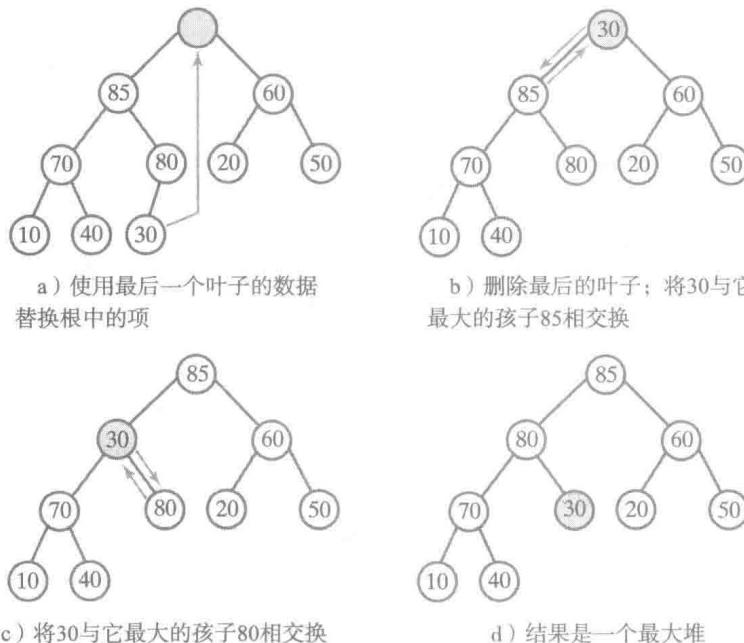


图 27-5 删除图 27-3d 所示最大堆根中的项的步骤

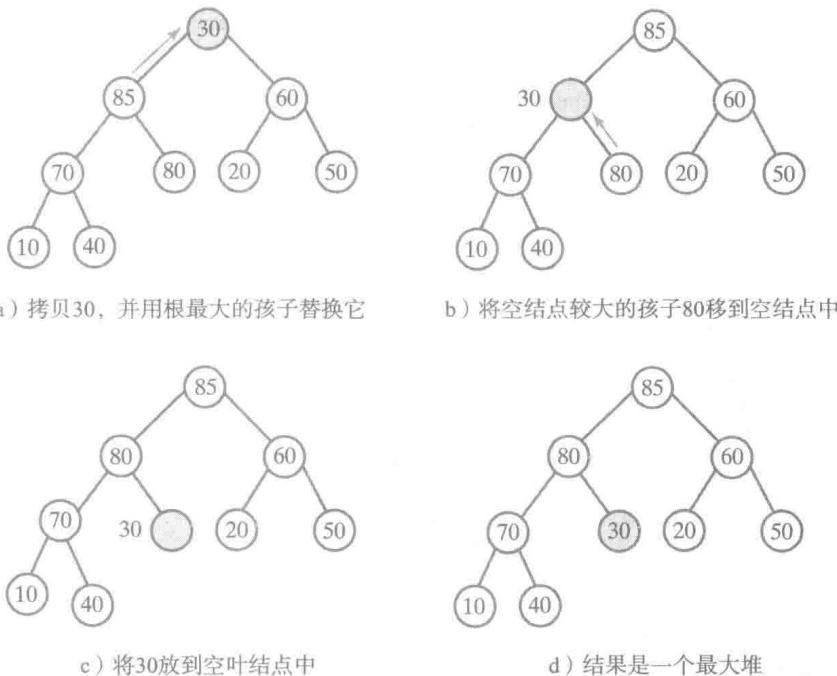


图 27-6 不需要交换将图 27-5b 所示的半堆转换为堆的步骤

下列算法将半堆转换为堆。为使算法能用于更一般的情况，假定半堆中的根位于给定的下标处而不是位置 1。

```

Algorithm reheap(rootIndex)
// Transforms the semiheap rooted at rootIndex into a heap
done = false
orphan = heap[rootIndex]
while (!done 且 heap[rootIndex] 有一个孩子)
{
    largerChildIndex = heap[rootIndex] 的较大孩子的下标
    if (orphan < heap[largerChildIndex])
    {
        heap[rootIndex] = heap[largerChildIndex]
        rootIndex = largerChildIndex
    }
    else
        done = true
}
heap[rootIndex] = orphan

```

如你所见，有几处会用到这个算法。



学习问题 7 跟踪算法 reheap 的步骤，显示对应于图 27-6 中各树的数组 heap 的内容。

方法 reheap。 reheap 算法实现为如下的私有方法。

```

private void reheap(int rootIndex)
{
    boolean done = false;
    T orphan = heap[rootIndex];
    int leftChildIndex = 2 * rootIndex;
    while (!done && (leftChildIndex <= lastIndex) )
    {

```

27.11

```

int largerChildIndex = leftChildIndex; // Assume larger
int rightChildIndex = leftChildIndex + 1;
if ( (rightChildIndex <= lastIndex) &&
    heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0)
{
    largerChildIndex = rightChildIndex;
} // end if
if (orphan.compareTo(heap[largerChildIndex]) < 0)
{
    heap[rootIndex] = heap[largerChildIndex];
    rootIndex = largerChildIndex;
    leftChildIndex = 2 * rootIndex;
}
else
    done = true;
} // end while
heap[rootIndex] = orphan;
} // end reheap

```

最差情况下，方法 `reheap` 沿从根到叶结点的路径执行。这条路径上结点的个数小于等于堆的高度 h 。所以 `reheap` 是 $O(h)$ 的。回忆一下，含 n 个结点的完全树的高度是 $\log_2(n+1)$ 向上取整。所以 `reheap` 方法是 $O(\log n)$ 的。

27.12 方法 `removeMax`。方法 `removeMax` 用最后一个叶结点替换堆的根，形成类似于图 27-6a 那样的半堆。然后方法调用 `reheap` 将半堆转回为堆。`removeMax` 的实现如下所示。

```

public T removeMax()
{
    checkIntegrity();           // Ensure initialization of data fields
    T root = null;
    if (!isEmpty())
    {
        root = heap[1];         // Return value
        heap[1] = heap[lastIndex]; // Form a semihheap
        lastIndex--;             // Decrease size
        reheap(1);               // Transform to a heap
    } // end if
    return root;
} // end removeMax

```

因为 `reheap` 在最差情况下是 $O(\log n)$ 的，故 `removeMax` 也是这样。

 **注：**为删除堆的根，先要用堆中最后的叶结点替换根。这个步骤得到一个半堆，因此使用方法 `reheap` 将半堆转换回堆。

创建堆

27.13 使用 `add`。可以由一个对象集合创建一个堆，方法是使用 `add` 方法，将每个对象添加到初始为空的堆中。图 27-7 显示用该方法将 20、40、30、10、90 和 70 添加到堆中的步骤。因为 `add` 是 $O(\log n)$ 的，故用这种方法创建堆将是 $O(n \log n)$ 的。

注意到，每次添加后都得到一个堆。这个过程超出了我们的需要。为了减少操作，从对象集合创建堆时，每个中间步骤可以不必保持堆形，下段将会介绍。

27.14 使用 `reheap`。更高效的创建堆的方法是使用方法 `reheap`。开始时，将要组成堆的项放到数组中从 1 开始的元素中。图 27-8a 提供了一个这样的示例数组。这个数组可以表示图 27-8b 所示的完全树。这棵树中含有可转为堆的半堆吗？叶结点是半堆，但它们也是堆。所以我们可以在忽视项 70、90 和 10。这些项位于数组的最后。

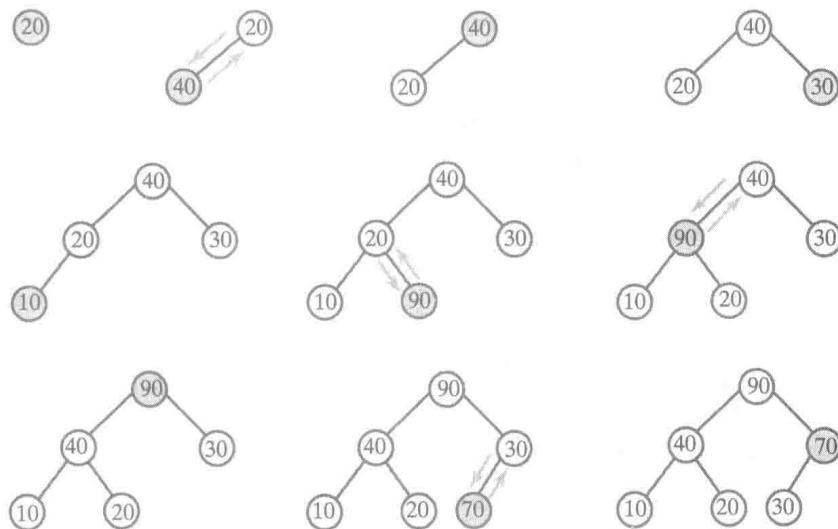
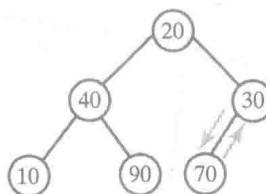


图 27-7 将 20、40、30、10、90 和 70 添加到初始为空的堆中的步骤

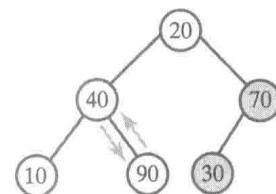
向数组头方向的下一个项是 30，这是图 27-8b 所示的树中一个半堆的根。如果将 `reheap` 应用于这个半堆，可得到图 27-8c 所示的树。继续这个办法，将 `reheap` 应用于以 40 为根的半堆，然后应用于以 20 为根的半堆。图 27-8d、27-8e 和 27-8f 显示这些步骤的结果。图 27-8f 是所需的堆。

| | | | | | | |
|---|----|----|----|----|----|----|
| | 20 | 40 | 30 | 10 | 90 | 70 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

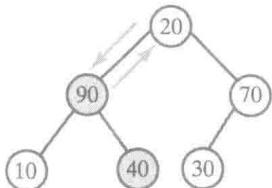
a) 保存项的数组



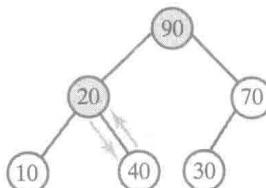
b) 数组表示的完全树



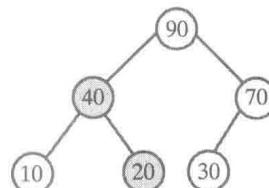
c) 执行 reheap(3)后



d) 执行 reheap(2)后



e) 执行 reheap(1)过程中



f) 执行 reheap(1)后

图 27-8 使用 `reheap` 创建由项 20、40、30、10、90 和 70 组成的堆的步骤

下列 Java 语句将数组 `heap`——其项位于下标 1 到 `lastIndex` 处——转为堆：

```
for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)
    reheap(rootIndex);
```

应用 `reheap` 时，从最靠近数组尾的第一个非叶结点开始。这个非叶结点在下标 `lastIndex/2` 处，因为它是树中最后一个叶结点的父结点。然后一直执行到 `heap[1]`。



学习问题 8 如果用数组表示一个堆，则 `lastIndex` 是堆中最后一个叶结点的下标，说明为什么最靠近数组尾的第一个非叶结点的下标是 `lastIndex/2`。

27.15 使用 `reheap` 将保存项的数组转为堆，比使用 `add` 将项添加到堆中，做的事情更少。事实上，以这种方式创建堆是 $O(n)$ 的，现在来说明这一点。

通过段 27.11 结尾处的观察，`reheap` 是 $O(h_i)$ 的，其中 h_i 是以下标 i 为根的子树的高度。最差情况下，堆将是高度为 h 的满树。层 $l < h$ 中的每个结点是其高度为 $h-l+1$ 的子树的根。另外，层 l 含有 2^{l-1} 个结点。故以满堆 l 层中结点为根的子树的高度之和是 $(h-l+1) \times 2^{l-1}$ 。

因为前一段中的循环中忽略了堆中最后一层——层 h ——中的结点，故它的复杂度是

$$O\left(\sum_{l=1}^{h-1} (h-l+1) \times 2^{l-1}\right)$$

本章结尾的练习 6 要求你说明这个表达式等价于 $O(2^h)$ ，这是 $O(n)$ 的。



注：可以使用方法 `reheap` 替代方法 `add`，创建堆的效率更高。

27.16 另一个构造方法。可以使用段 27.14 中描述的技术来实现类 `MaxHeap` 的另一个构造方法。假定要组成堆的 n 个项放在只有 n 个位置的数组中。下列构造方法将这个数组拷贝到数据域 `heap` 中，并使用 `reheap` 来创建堆。虽然给定数组中的项从下标 0 开始，但我们还是将其放到数组 `heap` 从下标 1 开始的元素中。注意，这个构造方法调用 `MaxHeap` 的第二个构造方法，所以要验证所需的容量，并分配数组 `heap`。

```
public MaxHeap(T[] entries)
{
    this(entries.length); // Call other constructor
    lastIndex = entries.length;
    // Assertion: integrityOK == true
    // Copy given array to data field
    for (int index = 0; index < entries.length; index++)
        heap[index + 1] = entries[index];
    // Create heap
    for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)
        reheap(rootIndex);
} // end constructor
```

堆排序

27.17 可以使用堆来排序一个数组。如果将数组项放到最大堆中，然后每次删除一个，则将得到降序排列的项。我们从段 27.13 和段 27.14 已经看到，从保存项的数组创建堆时，使用 `reheap` 比使用 `add` 的效率要高。实际上，在前一段所写的构造方法中，正是为此目的而调用的 `reheap`。所以如果 `myArray` 是项——例如字符串——的数组，就可以使用这个构造方法来创建堆，如下所示：

```
MaxHeapInterface<String> myHeap = new MaxHeap<>(myArray);
```

当从 `myHeap` 中删除项时，可以将它们倒着放回 `myArray` 中。这个问题的问题是需要额外的内存，因为堆在所给数组之外还使用了一个数组。但是，模仿堆的基于数组的实现，我们可以不使用类 `MaxHeap`，而提高了这个方法的效率。得到的算法称为堆排序（heap sort）。

27.18 要从给定的数组创建初始堆，可重复调用 `reheap`，正如段 27.16 中所给构造方法中的处理一样。图 27-9a 和图 27-9b 分别显示了一个数组及执行了这个步骤后得到的堆。因为数组要从下标 0 开始保存数据，但在构造方法中堆从下标 1 处开始，所以必须调整 `reheap`，

如你所见。

图 27-9b 所示数组中的最大项现在位于数组的第一个位置，所以将它与数组最后的项相交换，如图 27-9c 所示。数组现在分为树的部分和有序的部分。

交换后，在树的部分调用 `reheap`——将其转换为堆——并执行另一次交换，如图 27-9d 和图 27-9e 所示。重复这些操作直到树的部分只含有一个项时为止（图 27-9k）。数组现在按升序有序。注意，实际上数组在图 27-9g 时已有序，但算法并没有发现这个事实。

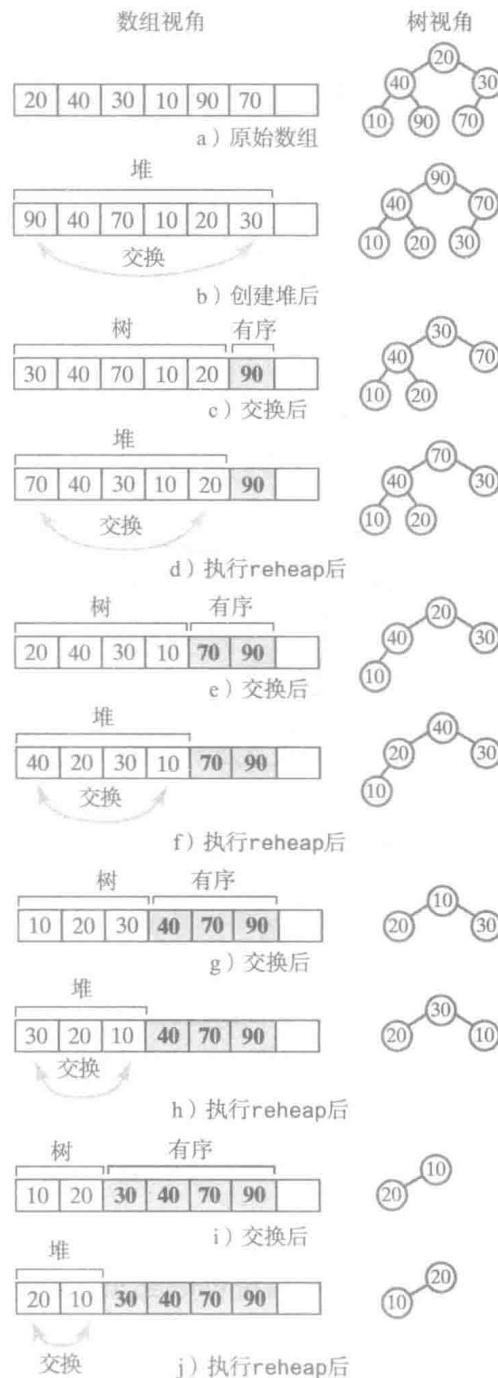


图 27-9 跟踪堆排序



图 27-9 (续)

27.19 调整 reheap。必须修改方法 reheap，以便它能适用于我们的排序算法。段 27.11 中的原始方法用到类 MaxHeap 中的数据域 heap 和 lastIndex。这里，让它们作为方法的参数。所以修改方法的头，如下所示。

```
private static <T extends Comparable<? super T>>
    void reheap(T[] heap, int rootIndex, int lastIndex)
```

数组 heap 中表示堆的部分，是从下标 0 到下标 lastIndex 处。半堆的根在下标 rootIndex 处。

因为堆从下标 0 而不是 1 开始，如段 27.11 中所述，故下标 i 处结点的左孩子在下标 $2i+1$ 而不是 $2i$ 处。回忆学习问题 1 要求你找到这个下标。这个改变影响了 reheap 中确定 leftChildIndex 值的两个语句。

修改后的 reheap 方法如下所示。

```
private static <T extends Comparable<? super T>>
    void reheap(T[] heap, int rootIndex, int lastIndex)
{
    boolean done = false;
    T orphan = heap[rootIndex];
    int leftChildIndex = 2 * rootIndex + 1;
    while (!done && (leftChildIndex <= lastIndex))
    {
        int largerChildIndex = leftChildIndex;
        int rightChildIndex = leftChildIndex + 1;
        if ((rightChildIndex <= lastIndex) &&
            heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0)
        {
            largerChildIndex = rightChildIndex;
        } // end if
        if (orphan.compareTo(heap[largerChildIndex]) < 0)
        {
            heap[rootIndex] = heap[largerChildIndex];
            rootIndex = largerChildIndex;
            leftChildIndex = 2 * rootIndex + 1;
        }
        else
            done = true;
    } // end while
    heap[rootIndex] = orphan;
} // end reheap
```

27.20 方法 heapSort。堆排序的实现从反复调用 reheap 开始，由给定的数组创建一个初始堆，如段 27.16 中所给的构造方法中的处理一样。但是，因为堆从下标 0 而不是 1 开始，所以必须调整循环：

```
for (int rootIndex = n / 2 - 1; rootIndex >= 0; rootIndex--)
    reheap(heap, rootIndex, n - 1);
```

这个循环假定 n 个项在数组 `heap` 中，从下标 0 开始。本章最后的练习 3 要求你验证 `rootIndex` 的开始值。

完整的方法如下所示。

```
public static <T extends Comparable<? super T>> void heapSort(T[] array, int n)
{
    // Create first heap
    for (int rootIndex = n / 2 - 1; rootIndex >= 0; rootIndex--)
        reheap(array, rootIndex, n - 1);
    swap(array, 0, n - 1);
    for (int lastIndex = n - 2; lastIndex > 0; lastIndex--)
    {
        reheap(array, 0, lastIndex);
        swap(array, 0, lastIndex);
    } // end for
} // end heapSort
```

与归并排序和快速排序一样，堆排序是 $O(n \log n)$ 的算法。此处所给的实现中，堆排序不需要第二个数组，但归并排序需要。回忆第 16 章，快速排序在大多数情况下是 $O(n \log n)$ 的，但在最差情况是 $O(n^2)$ 的。通常选择合适的枢轴可以避免快速排序的最差情况，故一般来讲，它是首选的排序方法。



注：堆排序的时间复杂度

虽然堆排序的平均情况是 $O(n \log n)$ 的，但是选择排序方法时常常选择快速排序。



学习问题 9 跟踪 `heapSort` 方法对数组 9 6 2 4 8 7 5 3 进行升序排序的步骤。

本章小结

- 因为堆是一棵完全二叉树，所以基于数组实现的效率高。
- 向堆中添加新项，将其作为完全二叉树中最后的叶结点。然后将项上浮到堆中合适的位置。
- 先用最后一个叶结点中的项来替代堆根中的项，然后再删除叶结点，从而删除了堆根中的项。结果为一个半堆。将新根中的项下沉到堆中合适的位置，从而将半堆转换为堆。
- 可以将给定的数组中的每个项添加到堆中，从而创建堆。更高效的方法是考虑数组表示的完全树，将每个非叶结点看作一个半堆。使用与删除堆根时使用的同样方法，将每个这样的半堆转化为一个堆。
- 堆排序使用堆来排序给定数组中的项。

练习

1. 使用段 27.16 中给出的构造方法，将下面每个数组构成最大堆，跟踪其过程。

a. 10 20 30 40 50

b. 10 20 30 40 50 60 70 80 90 100

2. 跟踪将下列各值依次添加到初始为空的最大堆的过程：

10 20 30 40 50

将这个过程与练习 1a 的过程相比较。

3. 段 27.20 中所给的方法 `heapSort` 中，包含了一个由含 n 个值的数组创建初始堆的循环。循环变量 `rootIndex` 的值从 $n/2-1$ 开始。推导这个起始值，并说明循环执行的次数与段 27.16 所给的构造方法中相应的循环执行次数相同。
4. 对下列每个数组，跟踪堆排序的过程：
 - a. 10 20 30 40 50 60
 - b. 60 50 40 30 20 10
 - c. 20 50 40 10 60 30
5. 考虑表示堆的数组。假定你用一个新值替换下标 i 处的值。很可能得到的不再是堆。写一个再次得到堆的算法。
6. 段 27.15 表明，使用 `reheap` 创建堆的复杂度是

$$O\left(\sum_{l=1}^{h-1} (h-l+1) \times 2^{l-1}\right)$$

说明这个表达式等价于 $O(2^h)$ ，这是 $O(n)$ 的。提示：首先，将加和变量从 l 改为 j ，其中 $j=h-l+1$ 。然后由归纳法证明：

$$\sum_{j=2}^h j/2^j = 3/2 - \frac{h+2}{2^h}$$

7. 考虑堆排序中由 n 个值的数组创建初始堆的循环（见段 27.20）：

```
// Create first heap
for (int rootIndex = n / 2 - 1; rootIndex >= 0; rootIndex--)
    reheap(array, rootIndex, n - 1);
```

说明执行这个循环过程中，方法 `compareTo` 的调用次数不少于 $n-1$ 。

8. 再次考虑前一练习中提到的循环。说明执行这个循环过程中，方法 `compareTo` 的调用次数不大于 $n \log n$ 。
9. 堆排序不是使用堆对数组排序的唯一方法。本练习要求你开发一个低效率的算法。建立初始堆后，与堆排序中第一步一样，最大值应该在数组的第一个位置。如果将这个值放在原地不动，用剩余值再建立一个新堆，将得到整个数组中第二大的值。继续这个过程，可以得到降序排序的数组。如果使用最小堆而不是最大堆，则得到升序排序的数组。
 - a. 实现其中一种排序。
 - b. 这个方法的大 O 性能是多少？

项目

1. 回忆第 24 章段 24.32，在最小堆中，每个结点中的对象小于等于结点后代中的对象。最大堆有方法 `getMax`，而最小堆有方法 `getMin`。使用数组实现最小堆。
2. 对随机选择的各种数组，比较堆排序、归并排序和快速排序的执行次数。第 4 章的项目中描述了对代码运行计时的一种方法。
3. 实现 `MaxHeapInterface` 时使用二叉查找树。树中哪个位置是最大的项？这种实现的效率如何？
4. 考虑将两个堆合并为一个堆的问题。
 - a. 写合并两个堆的高效算法，一个堆的大小是 n ，另一个是 l 。算法的大 O 性能是多少？
 - b. 写合并两个大小均为 n 的堆的高效算法。算法的大 O 性能是多少？
 - c. 写高效算法，将两个任意大小的堆合并为一个堆。算法的大 O 性能是多少？
 - d. 实现 c 问中的算法。
5. 通过下列步骤，可以研究堆排序中第一步——建立初始堆——的平均性能：

- 修改 `reheap` 方法，让它返回调用 `compareTo` 的次数。
- 写一个程序，将下列两个步骤执行 1000 次。
 1. 生成 n 个随机数，将它们放到一个数组中。
 2. 统计练习 7 所给的将数组转为堆的代码中所需的比较（调用 `compareTo`）次数。

计算每次迭代中的比较次数。循环结束后，将比较次数除以 1000，计算建立堆所需的平均比较次数。
- 在前一步中，令 $n=10、20、30、40、50、60、70、80、90、100、200、400$ 和 800 。对每个 n ，看看 `compareTo` 的平均调用次数是不是大于等于下限 $n-1$ （见练习 7），及小于等于上限 $n \log_2 n$ （见练习 8）。
- 6. 第 16 章项目 9 描述如何在含有 n 个值的集合中找到第 k 小的值，其中 $0 < k < n$ 。设计一个算法，使用最小堆在含 n 个值的集合中找到第 k 小的值。使用项目 1 中定义的最小堆的类，在客户层实现你的算法。
- 7. 考虑第 7 章项目 13b。
 - a. 如果使用最大堆替代队列 B，你认为会出现哪些不同？
 - b. 使用上述修改执行模拟，将结果与你预言的结果进行比较。
 - c. 将 b 中得到的结果，与为队列 B 使用优先队列的各种实现时得到的结果进行比较。
- 8. 重做前一个项目的 a 和 b，但使用最大堆替代双端队列 A，而不是替代队列 B。

平衡查找树

先修章节：第 24 章、第 25 章、第 26 章

目标

学习完本章后，应该能够

- 在添加后执行旋转来恢复 AVL 树的平衡
- 在 2-3 树中添加项或查找项
- 在 2-4 树中添加项或查找项
- 从给定的 2-4 树形成红黑树
- 在红黑树中添加项或查找项
- 描述 B 树的目的

在第 26 章看到，如果二叉查找树是平衡的，则树的操作是 $O(\log n)$ 的。不幸的是，添加和删除操作不能确保二叉查找树依然平衡。本章考虑能保持平衡的查找树，及它们的效率。

我们的目的是介绍几类平衡的查找树，并对它们进行比较。将讨论保持平衡的前提下项的添加算法。还将展示如何在树中进行查找。但不讨论项的删除算法，这个留给后续课程介绍。

树中的项通常都是对象，但为了画树时清晰简明，我们将项表示为整数。

AVL 树

28.1

第 24 章段 24.30 表明，对同一组数据可以形成几棵不同形状的二叉查找树。这些树中，有些是平衡的，有些不平衡。将一棵不平衡的二叉查找树，重排它的结点后可能得到一棵平衡的二叉查找树。回忆平衡二叉树中的每个结点，其子树的高度差不大于 1。

重排结点来维持树平衡的想法最早是由两位数学家 Adel'son-Vel'skii 和 Landis 在 1962 年提出的。以他们的名字命名的 AVL 树（AVL tree）是一棵当它不平衡时重排其结点的二叉查找树。仅当添加或删除一个结点时会扰乱二叉查找树的平衡。所以在这些操作时，AVL 树根据需要重排结点来保持它的平衡。

例如，图 28-1a、图 28-1b 和图 28-1c 展示了依次添加 60、50 和 20 后的二叉查找树。3 次添加后树不平衡了，不过，AVL 树将重排它的结点来恢复平衡，如图 28-1d 所示。这个重排称为右旋转（right rotation），因为你可以想象结点在结点 50 处旋转了。如果现在将 80 添加到树中，它仍保持平衡，如图 28-2a 所示。添加 90 打破了平衡性（图 28-2b），但左旋转（left rotation）可以恢复它（图 28-2c）。此时旋转作用在结点 80 处。一般地，如果结点 N 是旋转后子树的根，我们说在结点 N 处旋转。注意每次旋转后，树仍是二叉查找树。

讨论平衡时，我们有时会提到平衡结点（balanced node）。如果结点是一棵平衡树的根，即如果它两棵子树的高度差不大于 1，则结点是平衡的。

单旋转

28.2

右旋转。现在详细讨论前面提到的旋转。图 28-3a 显示了平衡的 AVL 树的一棵子树。

子树 T_1 、 T_2 和 T_3 的高度相同。在结点 C 的左子树 T_1 中的添加，将在 T_1 中增加一个叶结点。假定这样的一个添加使得 T_1 的高度加 1，如图 28-3b 所示。以结点 N 为根的子树不平衡了。

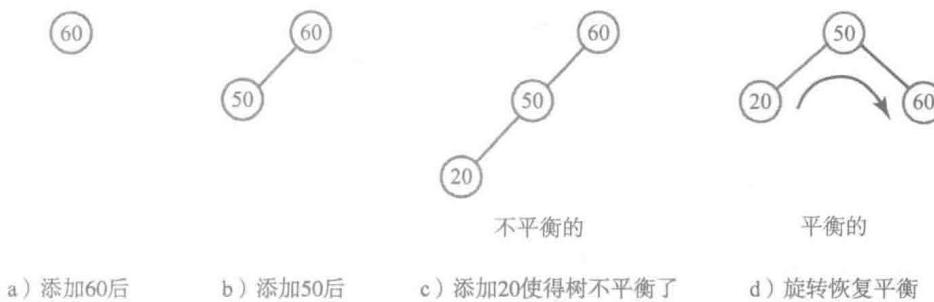


图 28-1 在初始为空的 AVL 树中进行添加

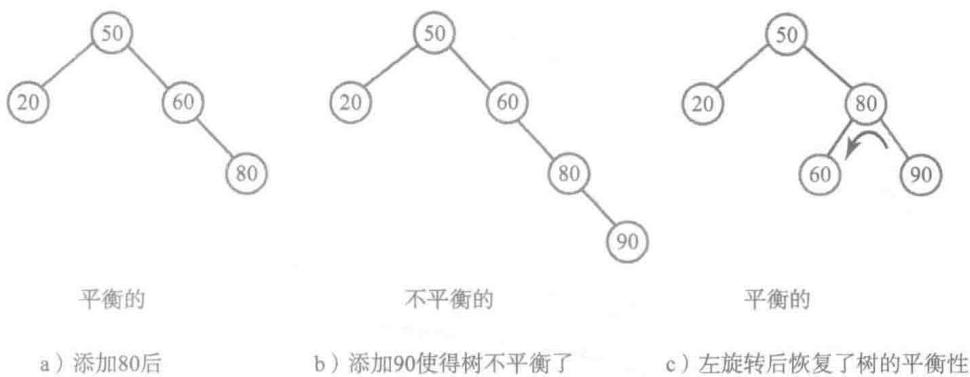


图 28-2 在图 28-1 所示的 AVL 树中添加

N 是从被插入的叶结点到 N 之间路径上第一个不平衡的结点。在结点 C 的右旋转恢复了树的平衡性，如图 28-3c 所示。旋转后， C 在 N 之上，树的高度与添加结点前的高度一样。

因为旋转前是二叉查找树（图 28-3b），故结点 N 中的值大于结点 C 中的值及 T_2 中的所有值。同时， T_2 中的所有值都大于结点 C 中的值。旋转后这些关系仍保持（图 28-3c），因为结点 N 是结点 C 的右孩子，而 T_2 是结点 N 的左子树。最后，子树 T_1 和 T_3 在新树中仍有原来的父结点。所以，得到的树仍是二叉查找树。

图 28-4 显示的是图 28-3 所描述的右旋转的具体示例。图 28-4a 所示为在树中插入 4 后结点 N 不平衡了。右旋转恢复了树的平衡性，如图 28-4b 所示。为了简化图的表示，我们仅标记出子树 T_1 、 T_2 和 T_3 的根结点。现在，结点 N 是 AVL 子树的根，而结点 C 成为根。如果结点 N 在旋转前有父结点，则旋转后让结点 C 成为那个父结点的孩子结点。

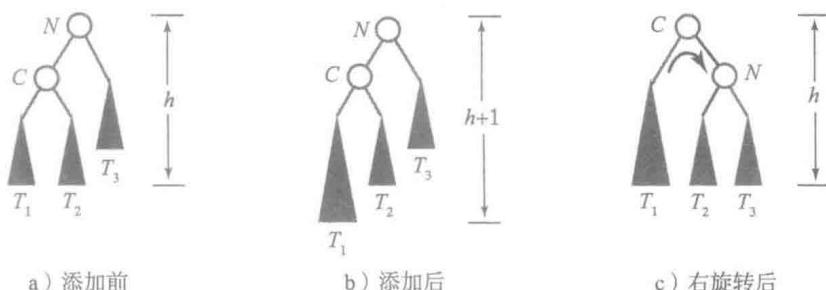


图 28-3 向 AVL 子树中添加之前和之后，需要右旋转来维持树的平衡

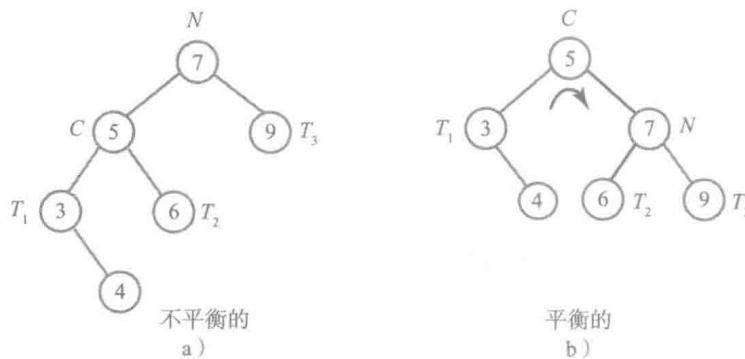


图 28-4 右旋转恢复 AVL 树的平衡之前和之后

下列算法执行图 28-3 和图 28-4 所示的右旋转。

```
Algorithm rotateRight (nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's left child.
```

```
nodeC=nodeN的左孩子
将nodeC的右孩子赋给nodeN的左孩子
将nodeN赋给nodeC的右孩子
return nodeC
```

学习问题 1 使用图 28-3 的标记法，标记图 28-1c 和图 28-1d 所示树的结点 N、C 及子树 T_1 、 T_2 和 T_3 。

28.3

左旋转。图 28-5 所示为图 28-3 镜像的左旋转。下列算法执行这个左旋转。

```
Algorithm rotateLeft (nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.
```

```
nodeC=nodeN的右孩子
将nodeC的左孩子赋给nodeN的右孩子
将nodeN赋给nodeC的左孩子
return nodeC
```

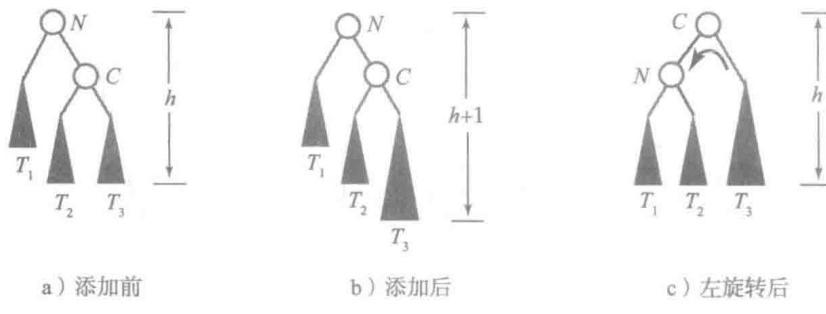


图 28-5 向 AVL 子树中添加之前和之后，需要左旋转来维持树的平衡

学习问题 2 为什么图 28-5c 中的树是二叉查找树？

学习问题 3 使用图 28-5 的标记法，标记图 28-2b、图 28-2c 所示树的结点 N、C 及子树 T_1 、 T_2 和 T_3 。

学习问题 4 就像图 28-4 给出右旋转示例一样，提供一个如图 28-5 所示的左旋转的具体示例。



注：因为向 AVL 树中插入而导致树中一个结点 N 的不平衡，可以通过单旋转来修正，如果

- 添加是在 N 的左孩子 C 的左子树中（右旋转），或是
 - 添加是在 N 的右孩子 C 的右子树中（左旋转）
- 这两种情况，都能推测出结点 C 旋转到结点 N 的上方。

双旋转

右-左双旋转。现在将 70 添加到图 28-2c 所示的 AVL 树中。不平衡点位于树根处，如图 28-6a 所示。在含 60 的结点处右旋转得到如图 28-6b 所示的树。这个旋转方式与图 28-3 所示的在结点 C 处的旋转一样。但这两个图中子树的高度不同。

不幸的是，这个旋转没能使树平衡。后面必须进行在含 60 结点处的左旋转——对应于图 28-5b 中的结点 C ——才能恢复平衡（图 28-6c）。这两个旋转一起称为右-左双旋转（right-left double rotation）。首先，60 旋转到 80 的上面，然后再旋转到 50 的上面。再重申一次，每次旋转，树仍是二叉查找树。

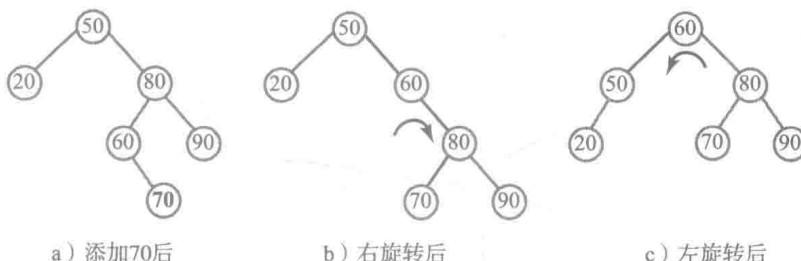


图 28-6 将 70 添加到图 28-2c 所示的 AVL 树中，需要一次右旋转和一次左旋转来保持树的平衡

现在来看一般的情形。图 28-7a 显示的是高度平衡的 AVL 树中的一棵子树。结点 N 有孩子 C 和孙子结点 G 。在结点 G 的右子树 T_3 中的添加，使 T_3 增加了一个叶结点。当这个添加操作增加了 T_3 的高度时，如图 28-7b 所示，以 N 为根的子树不平衡了。注意，结点 N 、 C 和 G 分别对应于图 28-6a 中含 50、80 和 60 的结点。

结点 N 是从被插入的叶结点到 N 之间的路径上第一个不平衡的结点。对结点 G 进行右旋转后，以 G 为根的子树不平衡了，如图 28-7c 所示。对 G 进行左旋转后恢复了树的平衡，如图 28-7d 中见到的。注意， G 先旋转到 C 的上面，然后是 N 的上面。

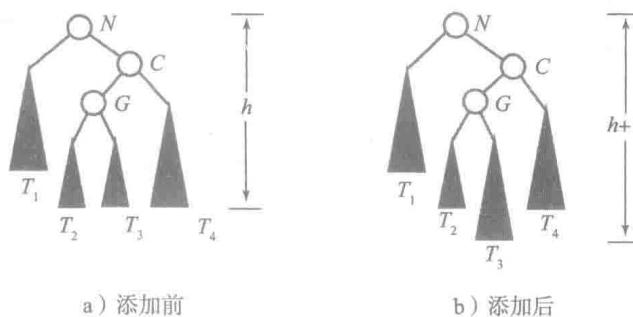


图 28-7 向 AVL 子树中添加之前和之后，需要一次右旋转和一次左旋转来保持树的平衡

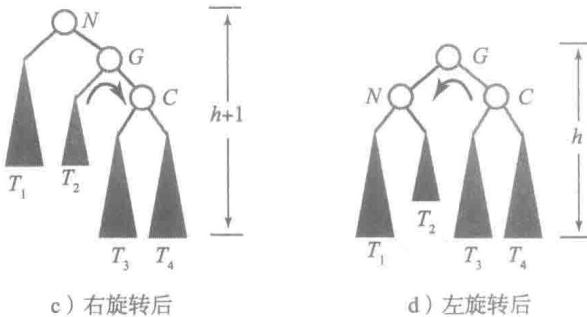


图 28-7 (续)

下列算法执行图 28-7 所示的右 - 左双旋转。

```

Algorithm rotateRightLeft (nodeN)
  // Corrects an imbalance at a given node nodeN due to an addition
   // in the left subtree of nodeN's right child.

```

nodeC=nodeN的右孩子
将rotateRight(nodeC)返回的结点赋给nodeN的右孩子
return rotateLeft(nodeN)

右旋转将图 28-7b 中的树转换为图 28-7c 中的树。然后左旋转将图 28-7c 中的树转换为图 28-7d 中的树。



学习问题 5 使用图 28-7 中的标记法，标记图 28-6 中的结点 N 、 C 和 G ，及子树 T_1 、 T_2 、 T_3 和 T_4 。

285

左-右双旋转。现在在图 28-6c 所示的树中添加 55、10 和 40，得到图 28-8a 所示的树。在添加 55 等这几个项时，添加仍保持了树的平衡性而不需要旋转。添加 35 后，树在含 50 的结点处不平衡，如图 28-8b 所示。要恢复平衡，对含 40 的结点执行左旋转——故 40 旋转到 20 的上面——得到图 28-8c 所示的树。然后对含 40 的结点执行右旋转——故 40 旋转到 50 的上面——得到图 28-8d 所示的树。

图 28-9 显示左 - 右双旋转的一般情况。它是图 28-7 所示的右 - 左双旋转的镜像。左 - 右双旋转和右 - 左双旋转都使得结点 G 先旋转到结点 C 的上面，

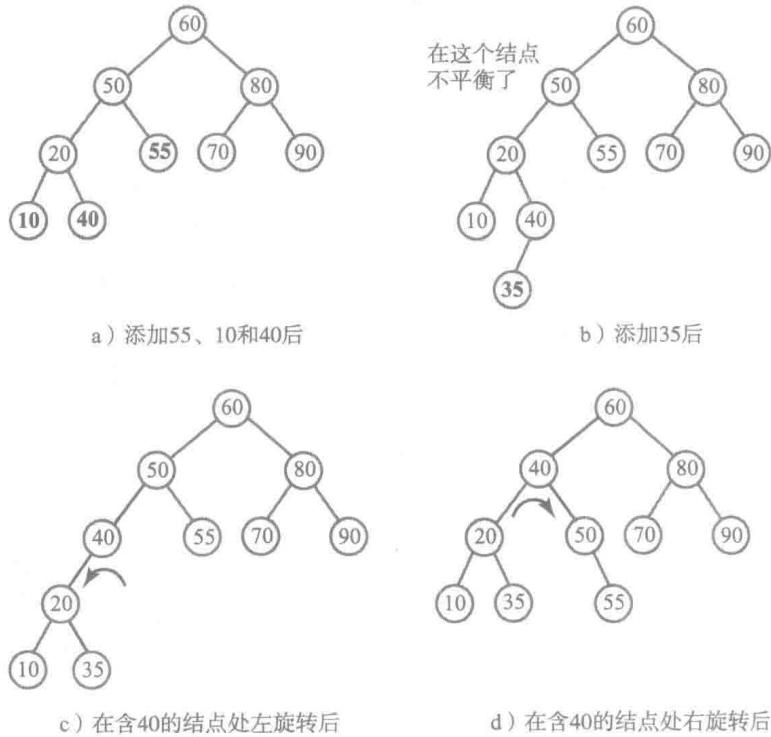


图 28-8 在图 28-6c 所示的 AVL 树中添加 55、10、40 和 35

然后是结点 N 的上面。

下列算法执行图 28-9 所示的左 - 右双旋转。

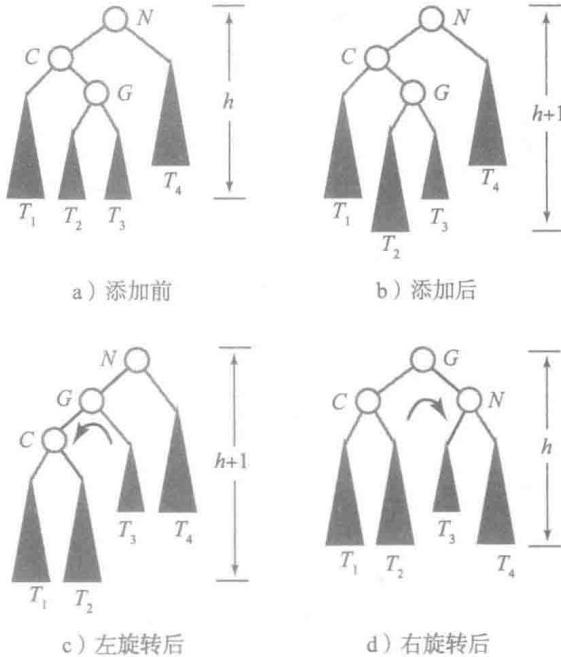


图 28-9 向 AVL 子树中添加之前和之后，需要左旋转和右旋转来保持树的平衡

Algorithm rotateLeftRight(nodeN)

// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's left child.

nodeC=nodeN的左孩子
将rotateLeft(nodeC)返回的结点赋给nodeN的左孩子
return rotateRight(nodeN)



学习问题 6 使用图 28-9 中的标记法，标记图 28-8 中的结点 N 、 C 和 G ，及子树 T_1 、 T_2 、 T_3 和 T_4 。



注：双旋转通过执行两次单旋转完成：

1. 在结点 N 的孙子结点（其孩子的孩子） G 处的旋转
2. 在结点 N 新孩子处的旋转

可以推测出 G 先旋转到 N 原来的孩子 C 的上方，然后是 N 的上方。



注：AVL 树中结点 N 的不平衡能通过双旋转来修正，如果

- 是在结点 N 右孩子的左子树上添加（右 - 左旋转），或是
- 是在结点 N 左孩子的右子树上添加（左 - 右旋转）

对添加后旋转操作的总结。每次添加到 AVL 树后，可能会出现暂时不平衡的现象。令 N 是最接近新叶结点的不平衡结点。单旋转或双旋转将恢复树的平衡性。不需要其他的旋转。要明白这一点，记得，添加前，树是平衡的；毕竟它是一棵 AVL 树。添加将导致一次旋转，树有与添加前同样的高度。所以，如果添加前是平衡的，那么在 N 之上就没有不

平衡的结点。而且，由以下的添加所引起的结点 N 不平衡的 4 种可能，都由这 4 种旋转解决了：

- 添加在结点 N 左孩子的左子树上（右旋转）
- 添加在结点 N 左孩子的右子树上（左 - 右旋转）
- 添加在结点 N 右孩子的左子树上（右 - 左旋转）
- 添加在结点 N 右孩子的右子树上（左旋转）

从二叉查找树中删除一项导致要删除一个结点，但不一定是删除包含这个项的结点。所以，从 AVL 树中删除一项可能导致暂时的不平衡。使用前面为添加操作所描述的单旋转或双旋转可以恢复树的平衡。将这些实现细节留作项目 1。

 注：添加项时的单旋转或双旋转将恢复 AVL 树的平衡。



学习问题 7 将下列项添加到初始为空的 AVL 树中，得到的树是什么？

70、80、90、20、10、50、60、40、30

学习问题 8 将前一题中所给的项添加到初始为空的二叉查找树中，得到的树是什么？将这棵树与前一题创建的 AVL 树进行比较。

学习问题 9 为什么图 28-7d 所示的树是一棵二叉查找树？

学习问题 10 为什么图 28-9d 所示的树是一棵二叉查找树？

28.7

AVL 树与二叉查找树。将 60、50、20、80、90、70、55、10、40 和 35 添加到初始为空的 AVL 树中，得到图 28-8d 所示的 AVL 树。图 28-10a 再次显示了这棵树。如果将同样的项添加到初始为空的二叉查找树中，得到图 28-10b 所示的树。这棵树是不平衡的，且比 AVL 树要高。

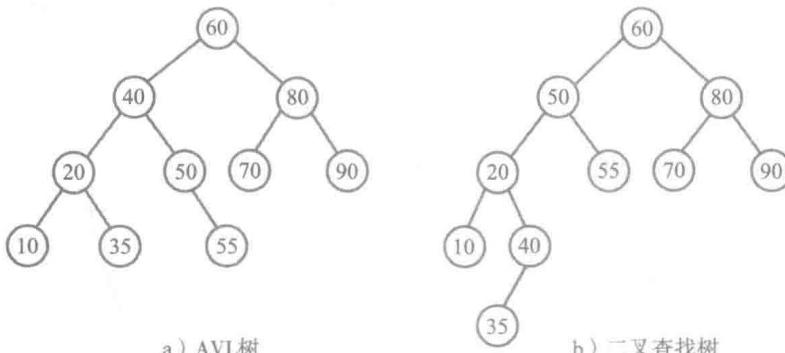


图 28-10 将 60、50、20、80、90、70、55、10、40 和 35 添加到初始为空的 AVL 树中和二叉查找树中

实现细节

28.8

类的框架。程序清单 28-1 概括了 AVL 树的类。因为 AVL 树也是一棵二叉查找树，所以我们从第 26 章讨论的 `BinarySearchTree` 类派生 `AVLTree` 类。方法 `add` 和 `remove` 很像是 `BinarySearchTree` 中的方法，但还需要一些逻辑规则，用来检测并修正可能出现的不平衡。所以我们需要重写这些方法。`SearchTreeInterface` 中规范说明的其他方法继承自 `BinarySearchTree` 类。

程序清单 28-1 AVLTree 类的框架

```

1 package TreePackage;
2 public class AVLTree<T extends Comparable<? super T>>
3     extends BinarySearchTree<T> implements SearchTreeInterface<T>
4 {
5     public AVLTree()
6     {
7         super();
8     } // end default constructor
9
10    public AVLTree(T rootEntry)
11    {
12        super(rootEntry);
13    } // end constructor
14
15    <Implementation of add and remove are here. A definition of add appears in Segment 28.12 of
16    this chapter. Other methods in SearchTreeInterface are inherited. >
17    . .
18    <Implementation of private methods to rebalance the tree using rotations are here. >
19    . .
20 } // end AVLTree

```

旋转。正如之前所讨论的，AVL 树在添加或删除一个结点后使用旋转来保持它的平衡性。执行这些旋转的方法严格遵从前一段给出的伪代码。28.9

例如，考虑段 28.2 中给出的单右旋转的算法。

Algorithm rotateRight (nodeN)
*// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's left child.*

nodeC=nodeN的左孩子
将nodeC的右孩子赋给nodeN的左孩子
将nodeN赋给nodeC的右孩子
return nodeC

下列方法将这个伪代码实现为 AVLTree 类中的私有方法。

```

// Corrects an imbalance at the node closest to a structural
// change in the left subtree of the node's left child.
// nodeN is a node, closest to the newly added leaf, at which
// an imbalance occurs and that has a left child.
private BinaryNode<T> rotateRight(BinaryNode<T> nodeN)
{
    BinaryNode<T> nodeC = nodeN.getLeftChild();
    nodeN.setLeftChild(nodeC.getRightChild());
    nodeC.setRightChild(nodeN);
    return nodeC;
} // end rotateRight

```

方法 rotateLeft 的实现与此类似，将其留作练习。

因为双旋转等价于两次单旋转，故执行双旋转的每个方法都调用执行单旋转的方法。例如，段 28.4 中出现的右 - 左双旋转算法如下。

Algorithm rotateRightLeft (nodeN)
*// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.*

nodeC=nodeN的右孩子
将rotateRight(nodeC)返回的结点赋给nodeN的右孩子
return rotateLeft(nodeN)

这个伪代码的实现如下。

```
// Corrects an imbalance at the node closest to a structural
// change in the left subtree of the node's right child.
// nodeN is a node, closest to the newly added leaf, at which
// an imbalance occurs and that has a right child.
private BinaryNode<T> rotateRightLeft(BinaryNode<T> nodeN)
{
    BinaryNode<T> nodeC = nodeN.getRightChild();
    nodeN.setRightChild(rotateRight(nodeC));
    return rotateLeft(nodeN);
} // end rotateRightLeft
```

方法 `rotateLeftRight` 的实现与此类似，将其留作练习。

学习问题 11 实现段 28.3 中给出的单左旋转算法。

28.10

再平衡。正如之前看到的，根据树形结构改变的位置，仅需要执行下列相应的旋转中的一种，就可以修正 AVL 树中因添加结点而导致的不平衡结点 N ：

- 如果添加在结点 N 左孩子的左子树上，执行右旋转
- 如果添加在结点 N 左孩子的右子树上，执行左–右旋转
- 如果添加在结点 N 右孩子的右子树上，执行左旋转
- 如果添加在结点 N 右孩子的左子树上，执行右–左旋转

下列伪代码使用这些准则及旋转方法使树再平衡。

```
Algorithm rebalance(nodeN)
if (nodeN的左子树的高度与其右子树的高度差大于1)
{
    // Addition was in nodeN's left subtree
    if (nodeN的左孩子的左子树高于其右子树)
        rotateRight(nodeN)      // Addition was in left subtree of left child
    else
        rotateLeftRight(nodeN) // Addition was in right subtree of left child
}
else if (nodeN的右子树的高度与其左子树的高度差大于1)
{
    // Addition was in nodeN's right subtree
    if (nodeN的右孩子的右子树比其左子树高)
        rotateLeft(nodeN)      // Addition was in right subtree of right child
    else
        rotateRightLeft(nodeN) // Addition was in left subtree of right child
}
```

如果结点 N 的两棵子树的高度相等或只差 1 则不再平衡。

28.11

rebalance 方法。返回结点左、右子树高度差的方法 `getHeightDifference`，将有助于我们实现前面的算法。给 `getHeightDifference` 方法返回的高度差加一个符号，这个方法就可以表示哪棵子树更高。这个方法可以定义在 `AVLTree` 类或是 `BinaryNode` 类中。如果每个结点都用一个或多个数据域来维护高度信息，而不是每次调用方法时重新进行计算，则后一种选择的效率更高。（见项目 4。）

如果结点的两棵子树的高度差大于 1，即如果 `getHeightDifference` 返回一个大于 1 或是小于 -1 的值，则结点是不平衡的。如果这个返回值大于 1，则左子树高；如果它小于 -1，则右子树高。

使用方法 `getHeightDifference`，可以实现前面 `AVLTree` 类中 `rebalance` 的伪代码。如下所示。

```

private BinaryNode<T> rebalance(BinaryNode<T> nodeN)
{
    int heightDifference = getHeightDifference(nodeN);

    if (heightDifference > 1)
    { // Left subtree is taller by more than 1,
      // so addition was in left subtree
      if (getHeightDifference(nodeN.getLeftChild()) > 0)
        // Addition was in left subtree of left child
        nodeN = rotateRight(nodeN);
      else
        // Addition was in right subtree of left child
        nodeN = rotateLeftRight(nodeN);
    }
    else if (heightDifference < -1)
    { // Right subtree is taller by more than 1,
      // so addition was in right subtree
      if (getHeightDifference(nodeN.getRightChild()) < 0)
        // Addition was in right subtree of right child
        nodeN = rotateLeft(nodeN);
      else
        // Addition was in left subtree of right child
        nodeN = rotateRightLeft(nodeN);
    } // end if
    // Else nodeN is balanced

    return nodeN;
} // end rebalance

```

add 方法。在 AVL 树中的添加，就像是在二叉查找树中的添加一样，只是再增加一个再平衡的步骤。例如，可以从 `BinarySearchTree` 中递归实现的 `add` 和 `addEntry` 方法（第 26 章段 26.15 和段 26.16）入手，修改它们，增加调用 `rebalance`。由此得到 `AVLTree` 中的方法，如下所示。

```

public T add(T newEntry)
{
    T result = null;

    if (isEmpty())
        setRootNode(new BinaryNode<>(newEntry));
    else
    {
        BinaryNode<T> rootNode = getRootNode();
        result = addEntry(rootNode, newEntry);
        setRootNode(rebalance(rootNode));
    } // end if

    return result;
} // end add

private T addEntry(BinaryNode<T> rootNode, T newEntry)
{
    //assert rootNode != null;
    T result = null;
    int comparison = newEntry.compareTo(rootNode.getData());
    if (comparison == 0)
    {
        result = rootNode.getData();
        rootNode.setData(newEntry);
    }

    else if (comparison < 0)
    {
        if (rootNode.hasLeftChild())
        {
            BinaryNode<T> leftChild = rootNode.getLeftChild();

```

28.12

```

        result = addEntry(leftChild, newEntry);
        rootNode.setLeftChild(rebalance(leftChild));
    }
    else
        rootNode.setLeftChild(new BinaryNode<T>(newEntry));
}
else
{
    //assert comparison > 0;
    if (rootNode.hasRightChild())
    {
        BinaryNode<T> rightChild = rootNode.getRightChild();
        result = addEntry(rightChild, newEntry);
        rootNode.setRightChild(rebalance(rightChild));
    }
    else
        rootNode.setRightChild(new BinaryNode<T>(newEntry));
} // end if

return result;
} // end addEntry

```

虽然在执行这个方法的过程中多次调用 `rebalance`, 但树的再平衡最多出现一次。对 `rebalance` 的大多数调用都仅仅是检查是否需要一次再平衡。

像 AVL 树一样吸引人的更好的查找树已经开发出来了, 稍后就会看到。

2-3 树

28.13 2-3 树 (2-3 tree) 是一棵一般查找树, 其内部结点必须含有 2 个或 3 个孩子。2- 结点 (2-node) 含有一个数据项 s 和两个孩子, 与二叉查找树中的结点一样。数据 s 大于结点左子树中的任意数据, 且小于右子树中的任意数据。即结点左子树中的数据小于 s , 而右子树中的数据大于 s , 如图 28-11a 所示。

3- 结点 (3-node) 含有两个数据项 s 和 l , 及 3 个孩子。小于较小数据项 s 的数据出现在结点的左子树中。大于较大数据项 l 的数据出现在结点的右子树中。介于 s 和 l 之间的数据出现在结点中间的子树中。图 28-11b 所示为一个典型的 3- 结点。

因为 2-3 树能含有 3- 结点, 所以它往往比二叉查找树更低。要使 2-3 树平衡, 需要所有的叶结点出现在同一层中。所以 2-3 树是一棵完全平衡树。

 注: 2-3 树是一棵一般查找树, 其内部结点必须含有 2 个或 3 个孩子, 且叶结点都在同一层中。2-3 树是一棵完全平衡树。

在 2-3 树中进行查找

28.14 如果有图 28-12 所示的一棵 2-3 树, 如何进行查找呢? 注意, 每个 2- 结点都遵循二叉查找树中的次序。3- 结点叶子 $<35\ 40>$ 中所含的值都介于其父

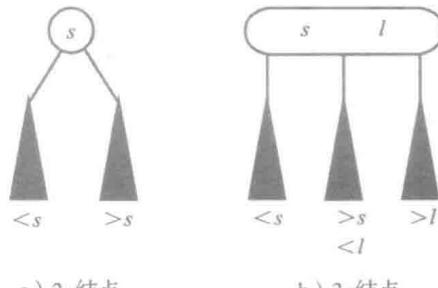


图 28-11 2-3 树中的结点

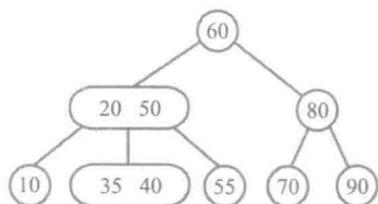


图 28-12 一棵 2-3 树

结点值之间。知道这一点后，就可以进行查找，例如查找 40，先将 40 与根的值 60 进行比较。然后转到 60 的左子树中，比较 40 与子树根中的值。因为 40 介于 20 和 50 之间，如果出现，它应该出现在中间那棵子树中。查找中间子树，比较 40 与 35，最后与 40 进行比较。

查找算法是二叉查找树查找算法的扩展。

```
Algorithm search23Tree(23Tree, desiredObject)
// Searches a 2-3 tree for a given object.
// Returns true if the object is found.

if (23Tree 为空)
    return false
else if (desiredObject 在 23Tree 的根中)
    return true
else if (23Tree 的根中含有两个项)
{
    if (desiredObject < 根中较小的对象)
        return search23Tree(23Tree 的左子树, desiredObject)
    else if (desiredObject > 根中较大的对象)
        return search23Tree(23Tree 的右子树, desiredObject)
    else
        return search23Tree(23Tree 的中间子树, desiredObject)
}
else if (desiredObject < 根中的对象)
    return search23Tree(23Tree 的左子树, desiredObject)
else
    return search23Tree(23Tree 的右子树, desiredObject)
```



学习问题 12 当在图 28-12 所示的 2-3 树中查找下列各值时，进行了哪些比较？

- a. 5 b. 55 c. 41 d. 30

向 2-3 树中添加项

结合示例，我们描述如何向 2-3 树中添加一个项。与在二叉查找树中的添加一样，2-3 树中也是将项添加到叶结点中。使用前一段所描述的查找算法找到这个叶结点。所以，一旦添加了，查找算法将能找到这个新项。

为了能将我们的结果与 AVL 树进行比较，我们选择与形成图 28-10a 所示的 AVL 树的序列相同的序列：60、50、20、80、90、70、55、10、40 和 35，插入初始为空的 2-3 树中。28.15

添加 60、50 和 20。添加 60 后，2-3 树含有唯一的 2- 结点。添加 50 后，树是一个 3- 结点。图 28-13a 和图 28-13b 分别显示了这两次添加后的树。28.16

现在添加 20。为了便于描述这次添加，在图 28-13c 中将 20 放在树的唯一的结点中。这是临时放置的地方，因为 3- 结点中只能含有两个数据项。实际上不能在这个结点中放更多的项。因为结点不能容纳 20，所以将它分裂 (split) 为 3 个结点，将中间值 50 提升一层。本例中，我们分裂的是叶结点同时也是树根结点。提升 50 需要创建一个新结点，它成为树的新根。这一步使得树高长 1，如图 28-13d 所示。



图 28-13 初始为空的 2-3 树，在三次添加后

28.17

添加 80、90 和 70。为了添加 80，注意到查找算法将在树的最右叶结点处查找 80。因为这个叶结点有空间放得下另一个数据项，所以那就是添加 80 的地方。图 28-14a 显示本次添加的结果。

查找算法将在刚添加了 80 的叶结点中查找 90。尽管叶结点中没有空间容纳另一个项，但我们想象将 90 添加在那里。然后将中间值——80——移高一层，将叶结点分裂为 60 和 90 的两个结点，如图 28-14b 所示。因为根可以接受 80，所以添加完成。

项 70 属于根的中间子树，因为这个叶子可以接受另一个项，所以将 70 添加在那里。图 28-14c 显示本次添加后的树。

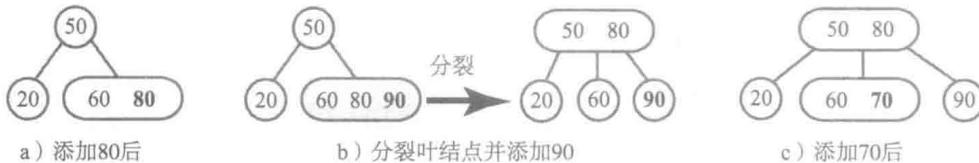


图 28-14 三次添加后的 2-3 树

28.18

添加 55。将 55 添加到图 28-14c 所示的树中时，查找算法在根的中间子树——叶结点——处终止，如图 28-15a 所示。因为这个叶结点不能容纳另一个项，所以分裂这个叶结点，60 提升到根中，如图 28-15b 所示。移动 60 导致根的分裂，60 提升为另一层中的新结点，它成为新的根。图 28-15c 显示本次添加后的树。

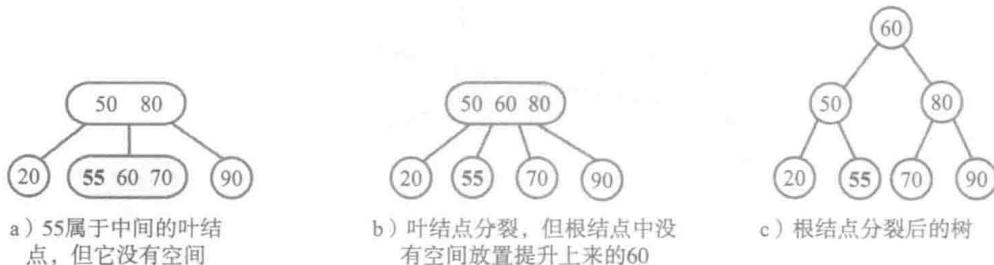


图 28-15 将 55 添加到图 28-14c 所示的 2-3 树中，导致叶结点然后是根结点的分裂

28.19

添加 10、40 和 35。图 28-15c 所示树中含 20 的叶结点，有空间接收 10 作为另一项，如图 28-16a 所示。另一个项 40，属于同一个叶结点，因为叶结点中已经含有两个项，所以将它分裂，将 20 提升一层，到含 50 的结点中。图 28-16b 和图 28-16c 显示了结果。最后，图 28-17 显示将 35 添加到树中的结果。含 40 的叶结点容纳了这个新项。

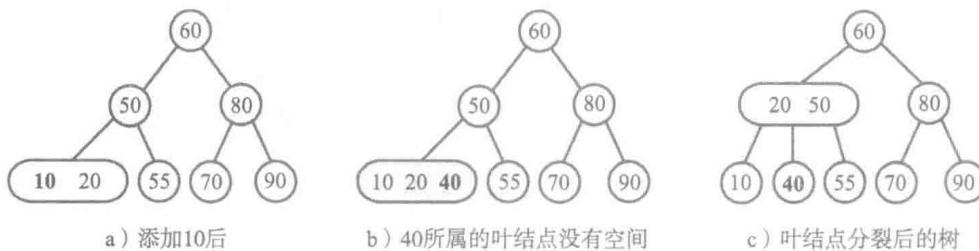


图 28-16 在图 28-15c 所示的 2-3 树中添加 10 和 40

比较图 28-17 中最终得到的 2-3 树与图 28-10a 中的 AVL 树。形成两棵树时使用的是相同的添加序列。2-3 树是完全平衡的，且低于平衡的 AVL 树。后面我们将这些树与下一节介

绍的 2-4 相比较，并得出一些结论。

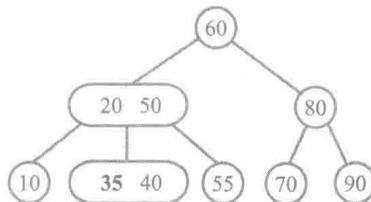


图 28-17 在图 28-16c 所示的 2-3 树中添加 35 后的树

添加过程中结点的分裂

叶结点的分裂。将新项添加到 2-3 树中时，第一个分裂的结点是已经含有两个项的叶结点。图 28-18a 显示了一个需要容纳 3 个项的叶结点。这些项按升序表示为 s 、 m 和 l : s 是结点中的最小项， m 是中间项，而 l 是最大项。结点分裂为分别含 s 和 l 的两个结点，而中间项 m 提升一层。如果叶结点的父结点有地方放 m ，则不需要进一步的动作。这是图 28-18a 中的情形。但在图 28-18b 中，父结点中已经含有两个项，所以也必须分裂它。接下来将讨论那种情形。

尽管图 28-18 显示叶结点是其父结点的右孩子，不过也可能是另一种类似的形态。

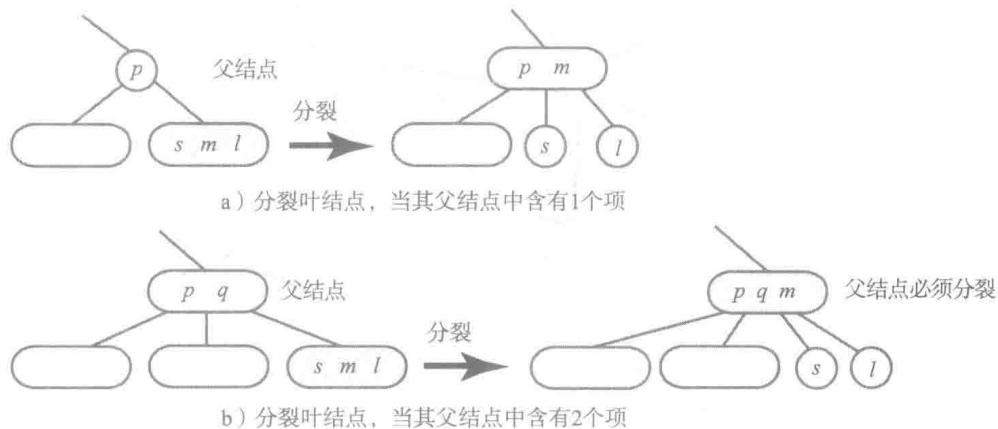


图 28-18 分裂叶结点来容纳新项

分裂内部结点。你刚刚看到，叶结点的分裂使得叶结点的父结点中含有太多的项。该父结点还有太多的孩子，如图 28-18b 所示。图 28-19 显示一般情况下这样的一个内部结点。这个结点必须容纳 3 个项，升序排列为 s 、 m 和 l ，及作为子树 T_1 到 T_4 的根的 4 个孩子。所以，分裂结点，将中间项 m 提升到结点的父结点中，将 s 和 l 放到各自的结点中，原结点的子树分配给 s 和 l 。如果父结点中有空间放 m ，则不需要进一步的分裂。如果没有，则父结点也像刚描述的这样进行分裂。

内部节点也有可能是其他类似的结构。

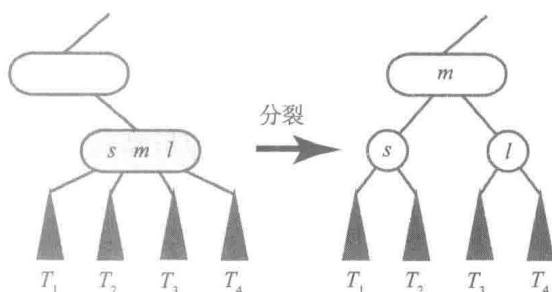


图 28-19 分裂一个内部结点来容纳新项

28.20

28.21

28.22

分裂根。根的分裂过程类似于前面这些情形，不同的是，将项提升一层时，要为该项分配一个新结点。这个新结点成为树的根，如图 28-20 所示。注意，这是 2-3 树高度增加的唯一情形。

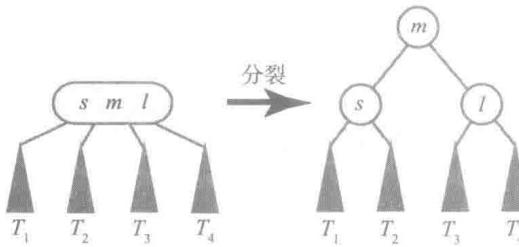


图 28-20 分裂根结点来容纳新项



学习问题 13 向图 28-17 所示的 2-3 树中添加 30 后得到的树是什么？

学习问题 14 将下列项添加到初始为空的 2-3 树中时得到的树是什么？

70、80、90、20、10、50、60、40、30

学习问题 15 前一题中创建的树，与学习问题 7 中创建的 AVL 树相比较，结果如何？

2-4 树

28.23

2-4 树 (2-4 tree) 有时也称为 2-3-4 树 (2-3-4 tree)，是一棵一般查找树，其内部结点必须含有 2、3 或 4 个孩子，且其叶结点要在同一层中。除我们在前一节中描述的 2- 结点和 3- 结点外，这种树中还含有 4- 结点。4- 结点 (4-node) 含有 3 个数据项 s 、 m 和 l ，并有 4 个孩子。小于最小数据项 s 的数据出现在结点的左子树中。大于最大数据项 l 的数据出现在结点的右子树中。介于 s 和中间数据项 m 之间的数据或介于 m 和 l 之间的数据，分别出现在结点中间的两棵子树中。图 28-21 所示为一个典型的 4- 结点。



注：2-4 树是一棵一般查找树，其内部结点必须含有 2、3 或 4 个孩子，且其叶结点要在同一层中。2-4 树是一棵完全平衡树。

除了多出来的处理 4- 结点的逻辑外，2-4 树中的查找与 2-3 树中的查找是一样的。这个查找是向 2-4 树中添加项的算法的基础。

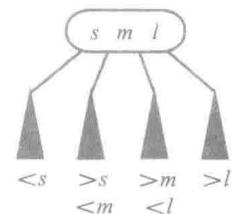


图 28-21 一个 4- 结点

向 2-4 树中添加项

28.24

回忆如何将新项添加到 2-3 树中。我们沿着从根结点开始到叶结点结束的一条路径进行比较。到叶结点时，如果叶结点是 3- 结点，则它已经含有两个数据项，所以必须分裂它。因为此时一个项会提升一层，所以这个分裂可能需要分裂叶结点之上的结点。故向 2-3 树中的添加可能需要我们沿路径从叶结点又折回到根。

在 2-4 树中，在从根结点到叶结点的查找过程中，一旦发现 4- 结点就分裂它，从而避免了 2-3 树中的折回。分裂后，比较路径上的下一个结点是分裂后的结果结点，所以它不是 4- 结点。如果这个结点有一个孩子是 4- 结点，且是我们下一步要处理的，则该结点有空间能容纳从孩子提升上来的项。不会再出现像 2-3 树中发生的那种再次分裂。你马上就会看到

一个示例。

与前一节一样，我们使用示例来说明如何向 2-4 树中添加项。我们使用之前用过的相同的添加序列——即 60、50、20、80、90、70、55、10、40 和 35——为的是可以将结果与前面的树进行比较。

添加 60、50 和 20。**图 28-22** 显示添加 60、50 和 20 到初始为空的 2-4 树中的结果。得到的树含有唯一的 4- 结点。**28.25**



图 28-22 添加 60、50 和 20 到初始为空的 2-4 树中

添加 80 和 90。为了将项添加到图 28-22c 所示的 2-4 树中，我们发现根是一个 4- 结点。**28.26** 分裂它，将中间项 50 提升。因为已经处于根了，所以为 50 创建一个新结点。那个结点成为树的新根，如图 28-23a 所示。现在可以将 80 和 90 添加到根的右叶子中了，如图 28-23b 和图 28-23c 所示。

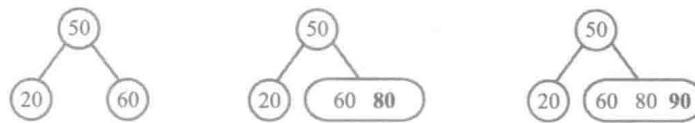


图 28-23 在图 28-22c 所示的树中添加 80 和 90

添加 70。在图 28-23c 所示的 2-4 树中查找添加 70 的位置时，碰到根的右孩子是一个 4- 结点。将这个结点分裂为两个结点，中间项 80 提升到根中。这次分裂的结果如图 28-24a 所示。现在根的中间孩子有空间容纳 70 了，如图 28-24b 所示。

添加 55、10 和 40。图 28-24b 所示的 2-4

树可以容纳 55、10 和 40 的添加，而不需要分裂结点。这些添加的结果如图 28-25 所示。

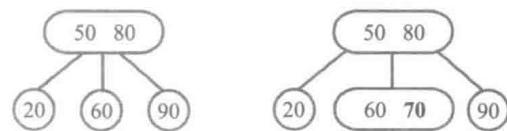


图 28-24 在图 28-23c 所示的 2-4 树中添加 70

28.27

28.28

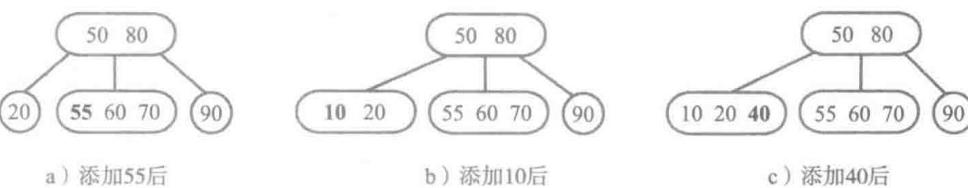


图 28-25 在图 28-24b 所示的 2-4 树中添加 55、10 和 40

添加 35。在图 28-25 所示的 2-4 树中添加 35 时，查找过程遇到了根的左孩子，这是一个 4- 结点。将这个结点分裂为两个结点，中间项 20 提升到根中，如图 28-26a 所示。现在可以将 35 添加到根的中左孩子中了，如图 28-26b 所示。这是我们要做的最后一次添加。

28.29



a) 在为35查找空间时分裂遇到的4-结点叶子后

b) 添加35后

图 28-26 在图 28-25c 所示的 2-4 树中添加 35



注: 当在 2-4 树中添加新项时, 为新项在树中查找位置的过程中, 一旦遇到 4- 结点就分裂它。查找结束后添加即完成。所以在 2-4 树中的添加比 2-3 树中的添加效率高。

学习问题 16 当在图 28-26b 所示的 2-4 树中查找下列各值时进行的比较是什么?

- a. 5 b. 56 c. 41 d. 30

学习问题 17 当将 30 添加到图 28-26b 所示的 2-4 树中时, 得到的结果是什么?

学习问题 18 将下列各项添加到初始为空的 2-4 树时得到的 2-4 树是什么?

70、80、90、20、10、50、60、40、30

学习问题 19 前一题中创建的树, 与学习问题 14 中创建的 2-3 树相比较, 结果如何?

AVL 树、2-3 树和 2-4 树的比较

28.30

图 28-27 比较了图 28-10a 中的 AVL 树、图 28-17 中最终的 2-3 树和刚刚构造的 2-4 树。AVL 树是高度为 4 的平衡二叉查找树。另外的两棵树是完全平衡的一般查找树。2-3 树的高度是 3; 2-4 树的高度是 2。一般地, 2-4 树比 2-3 树更低些, 2-3 树又比 AVL 树更低些。

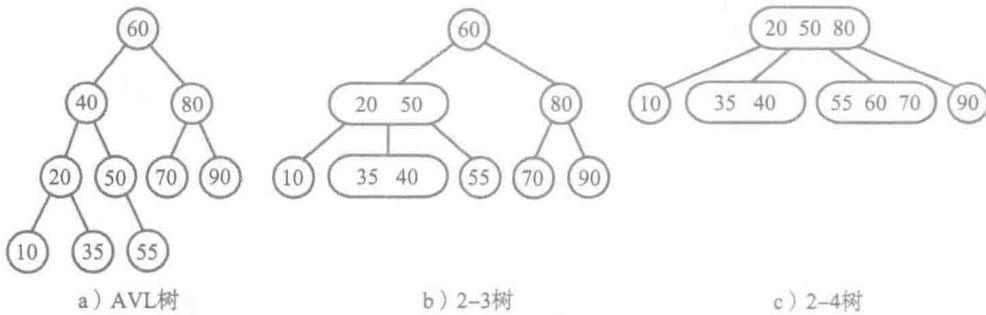


图 28-27 添加 60、50、20、80、90、70、55、10、40 和 35 后得到的三棵平衡查找树

第 26 章段 26.41 中已经看到, 在如 AVL 树这样的一棵平衡的二叉查找树中的查找是 $O(\log n)$ 的操作。因为 2-3 树和 2-4 树不高于对应的 AVL 树, 故在树中查找时检查的结点数通常会更少。但是, 3- 结点和 4- 结点比 2- 结点含有更多的项, 所以它们需要更长的查找时间。一般地, AVL 树、2-3 树或是 2-4 树的查找都是 $O(\log n)$ 的操作。

2-3 树是受人关注的, 因为维持它的平衡性比 AVL 树要容易。维持 2-4 树的平衡性甚至更容易些。但定义查找树中的结点含有多于 3 个的数据项时, 效果通常适得其反, 因为如果你进行顺序查找的话, 每个结点内比较的次数增多了。不过如果在数组中有很多的有序数据项, 则可以使用二分查找。这个查找最多是 $O(\log n)$ 的。本章后面你会看到, 这样的一棵查找树当将它保存在磁盘这样的外部存储而不是内存时, 还是有吸引力的。这些树是 B 树, 我们将在段 28.40 讨论。

红黑树

刚才提到，维持 2-4 树的平衡性比维持 AVL 树或是 2-3 树的平衡性都要容易些。而 2-4 树是一棵一般树，红黑树（red-black tree）是与 2-4 树等价的二叉树。向红黑树中添加项很像是向 2-4 树中的添加，过程中仅需要从根结点到叶结点走一趟。但红黑树是一棵二叉树，所以它使用比 2-4 树更简单的操作来维持其平衡性。另外，实现红黑树时仅用到 2- 结点，而 2-4 树需要 2- 结点、3- 结点和 4- 结点。2-4 树中的添加条件使得它不如红黑树更让人满意。

 注：红黑树是与 2-4 树等价的一棵二叉树。从概念上来讲，红黑树比 2-4 树更难懂，但它的实现仅用到 2- 结点，所以更容易实现。

设计 2-4 树中的结点时，必须考虑如何表示结点中的项。因为这些项必须有序，所以可以使用有序线性表这样的 ADT 来表示项。也或许会使用二叉查找树。例如考虑图 28-27c 中的 2-4 树。树根中的项是 20、50 和 80。可以用二叉查找树来表示这些项，树根为 50，其子树是 20 和 80。同样，这棵 2-4 树中 3- 结点叶结点中的项是 35 和 40。可以用两棵二叉查找树之一来表示这些项：一棵树根为 35，40 为其右子树；另一棵树根为 40，35 是其左子树。所以，可以将所有 3- 结点和 4- 结点都转换为 2- 结点。得到的是替代 2-4 树的二叉查找树。

每次将 3- 结点或 4- 结点转换为 2- 结点时，都会增加树的高度。我们使用颜色来标记导致高度增加的这些新结点。原 2-4 树中的所有结点使用黑色。因为没有改变 2- 结点，所以在新树中它们依然是黑色的。

图 28-28a 显示如何使用 2- 结点来表示 4- 结点。得到的子树的根保持黑色，但它的孩子结点涂了颜色。传统上的颜色是红色。我们的图使用白色代替红色。类似地，图 28-28b 显示如何使用两棵不同的子树之一来表示 3- 结点，每一棵都有一个黑根结点和一个红色的孩子结点。

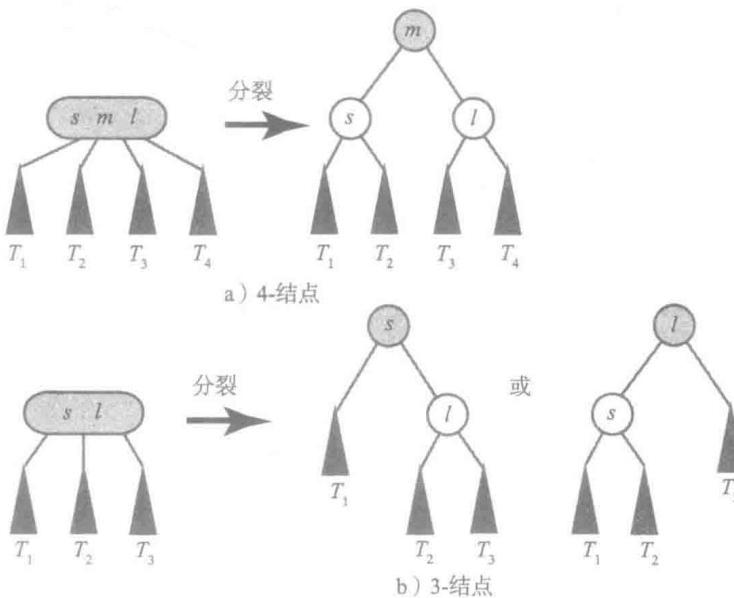


图 28-28 使用 2- 结点来表示

使用这种标记法，可以将图 28-27c 中的 2-4 树，画为图 28-29 中的平衡二叉查找树。这棵二叉查找树称为红黑树。



学习问题 20 在图 28-27c 所示的 2-4 树中，及图 28-29 所示的等价的红黑树中，找出下列项时进行的比较是什么？

- a. 60 b. 55

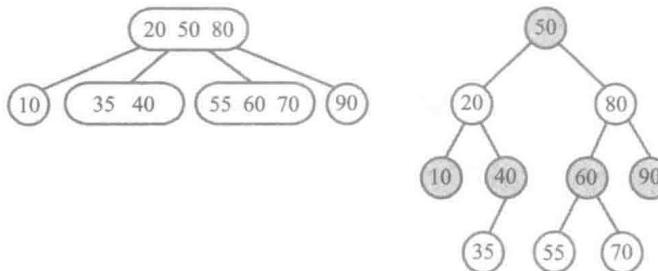


图 28-29 2-4 树（图 28-27c）及与之等价的红黑树

红黑树的特性

28.33 每棵红黑树的根都是黑色的。如果原 2-4 树的根是 2- 结点，则 2- 结点将是黑色的。而如果它的根是 3- 结点或 4- 结点，则将用一棵根为黑色的子树来替换它，如图 28-28 所示。

因为仅当将 3- 结点和 4- 结点转换为 2- 结点时才创建红色结点，所以每个红色结点都有一个黑色的父结点，可见于图 28-29 中。由此得出，红色结点不能有红色的孩子结点。如果有，则红色的孩子将有红色的父结点，这与前面每个红色结点都有一个黑色父结点的结论相矛盾。

当产生与 2-4 树等价的红黑树时，2- 结点仍为黑色，表示其他的任何结点时都含有一个黑色结点。所以 2-4 树中的每个结点都只在等价的红黑树中产生一个黑结点。因为 2-4 树是完全平衡树，故从根结点到叶结点的所有路径都连接了同样多的结点。所以红黑树中每条从根结点到叶结点的路径中都必须含有相同个数的黑结点。



注：红黑树特性

1. 根是黑色的。
2. 每个红结点都有一个黑色父结点。
3. 红结点的孩子都是黑色的，即红结点不能有红色的孩子结点。
4. 从根到叶结点的每条路径中都含有相同个数的黑结点。



学习问题 21 说明图 28-29 所示的红黑树满足刚提到的 4 个特性。

学习问题 22 与图 28-25c 中的 2-4 树等价的红黑树是什么？

学习问题 23 说明学习问题 22 得到的红黑树满足前面给出的 4 个特性。



注：创建一棵红黑树

实际上，不会将 2-4 树转换为红黑树。而是根据下一节描述的步骤，通过将项添加到初始为空的红黑树中来创建红黑树的。这些步骤既考虑了树的平衡性，也考虑了结点的颜色。

向红黑树中添加项

28.34

添加叶结点。添加到红黑树中的新结点应该指定什么颜色呢？如果树是空的，新结点将是树的根结点，所以必须是黑色的。非空二叉查找树的添加永远是在叶结点处，这对红黑树

也是成立的。如果新的叶结点使用黑色，将增加从根结点到叶结点的路径中黑结点的个数。这样的增加违反了红黑树的 4 个特性。所以添加到非空红黑树中的新结点必须是红色的。但是，不能认为红黑树中的所有叶结点都是红色的。添加或删除项可能改变个别结点的颜色，包括刚刚添加的叶结点。

注：添加到红黑树中的结点的颜色

如果向空红黑树中添加一个结点，则结点必须是黑色的，因为它是根。向非空红黑树中添加项，则得到一个红色的新叶结点。后面当有其他项添加或删除时，这个叶结点的颜色可能会改变。

考虑向红黑树中添加的一些简单示例。单结点的红黑树有一个黑结点，这个结点是它的根。图 28-30 显示当向这棵树中添加新项 e 时的两种可能。每种情形中，新的红结点都保持红黑树的特性，所以是合法的。

现在假定，添加新项 e 前红黑树中有两个结点。图 28-31a 显示了这棵原始树，此时它含有根 x 和右孩子 y 。另

一个图是等价于原红黑树的 2-4 树。其他的图显示的是，根据 e 与 x 和 y 相比较导致的可能添加结果。在图 28-31b 中， e 是根的左孩子，添加完成。在图 28-31c 中，红结点有红色的孩子结点。这两个连续的红结点在红黑树中是非法的（特性 2 和性质 3）。为了理解还需要进行哪些动作，考虑等价的 2-4 树。原来含 2 个结点的红黑树等价于含单结点 $\langle x \ y \rangle$ 的 2-4 树（图 28-31a）。如果添加大于 y 的项 e ，则 2-4 树变成单结点 $\langle x \ y \ e \rangle$ 树（图 28-31c）。注意到等价于它的红黑树含 3 个结点。这棵树是我们添加 e 后所需的结果树。我们可以将图 28-31c 中的第一棵红黑树在含 y 的结点处执行一次单左旋转而得到它。以前介绍 AVL 树时在图 28-5b 和图 28-5c 中见过这个旋转。旋转后，必须将含 x 和 y 的结点——即新结点原来的父结点和祖父结点——的颜色互换。我们称这一步为颜色翻转（color flip）。

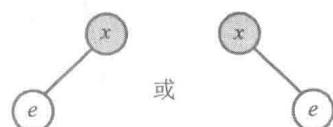


图 28-30 向单结点的红黑树中
添加新项 e 的结果

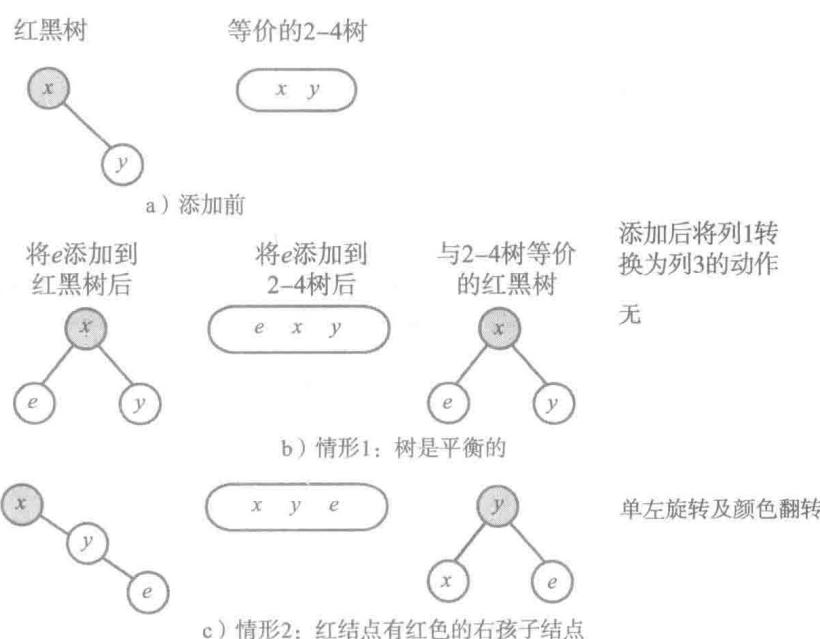
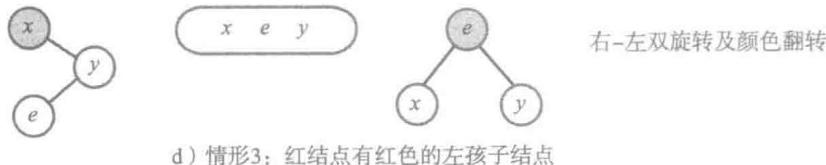


图 28-31 将新项 e 添加到含两个结点的红黑树中的可能结果

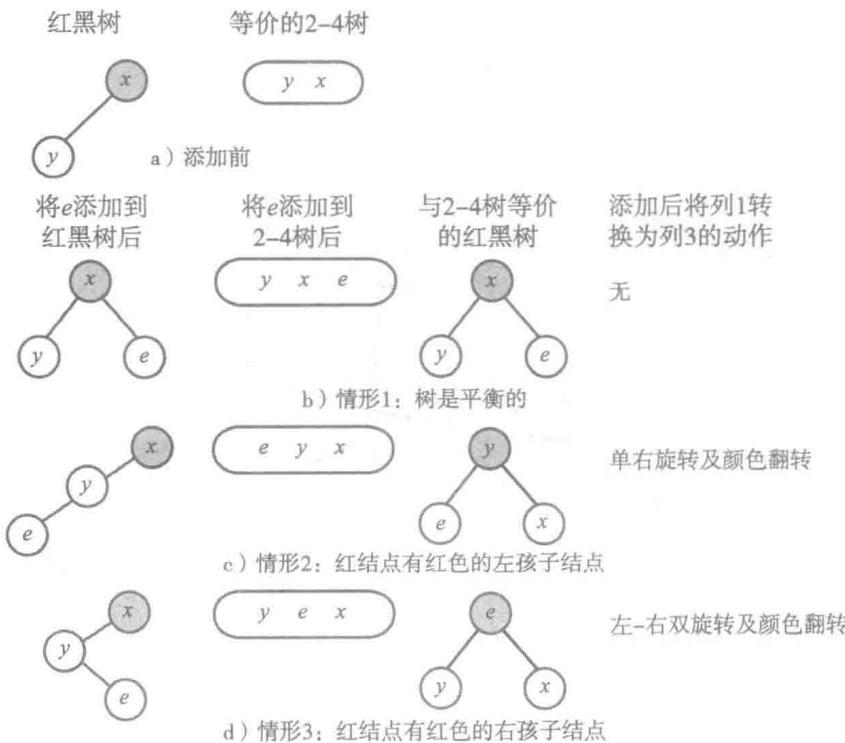


d) 情形3：红结点有红色的左孩子结点

图 28-31 (续)

图 28-31d 显示的是将 e 添加到两个结点的红黑树中最后一种可能的结果。这种情况下，为了避免两个连续的红结点，必须进行右 - 左双旋转，随后还要将新结点和它原来的祖父结点的颜色翻转。图 28-7b、图 28-7c 和图 28-7d 展示了在 AVL 树中一般情况下的旋转。

图 28-32 所示为图 28-31 中所示情形的镜像。

图 28-32 将新项 e 添加到含两个结点的红黑树中的可能结果：图 28-31 的镜像

28.35

父结点为黑结点的 4- 结点的分裂。在 2-4 树的添加过程中，沿从根到最终插入点之间的路径移动时，分裂遇到的任何 4- 结点。在红黑树中添加时必须执行等价的动作。图 28-28a 显示当黑结点有两个红孩子结点时，我们遇到的就是 4- 结点的红 - 黑表示。我们将这种结构称为红黑 4- 结点，或简称为 4- 结点。



注：红黑 4- 结点

一个红黑 4- 结点含有一个黑结点和两个红色孩子结点。

图 28-33a 让我们回忆起，在 2-4 树中，当 4- 结点的父结点是 2- 结点时如何分裂它。中间项 m 提升到结点的父结点中，其他的项 s 和 l 自成结点替代为父结点的孩子结点。图 28-33b 显示的是对应的红黑树。注意到，以 m 为根的子树的 3 个结点翻转了颜色。所以，通过颜色翻转分裂了 4- 结点的红黑表示。

当红黑 4- 结点有一个黑色父结点时，颜色翻转就是必须要做的全部事情。正如你在图 28-33 中所看到的，黑色父结点对应于 2-4 树中的 2- 结点。如果 2-4 树中的 4- 结点的父结点是一个 3- 结点，则红黑 4- 结点将有红色的父结点。下一段将讨论这种情况。

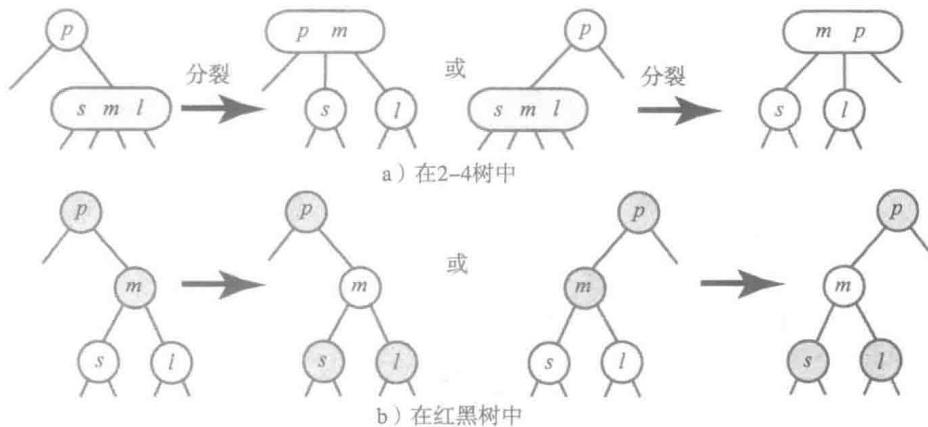


图 28-33 其父结点是 2- 结点的 4- 结点的分裂

父结点为红结点的 4- 结点的分裂：情形 1。 图 28-34a 显示了 2-4 树中，其父结点为 3- 结点的 4- 结点的分裂。28.36 这里，4- 结点是其父结点的右孩子。图 28-34b 显示的是图 28-34a 中两棵树的红黑树表示。我们如何将第一棵红黑树转换为第二棵？图 28-35 显示了必需的步骤。在图 28-35a 中，检测到在 m 处是 4- 结点，因为这个黑结点有两个红色的孩子结点。颜色翻转得到两个相邻的红结点，如图 28-35b 所示。之前在图 28-31c 中，我们见过这种一个黑结点带右侧两个连续子孙红结点的结构。与那时的处理一样，在 p 结点处执行左旋转，如图 28-35c 所示，然后翻转含 p 和 g 的结点的颜色。颜色翻转及旋转一起，解决了红结点的不合法性。图 28-35d 中的结果就是我们在图 28-34b 中见过的所需的红黑树。

因为 3- 结点有两种不同的红黑树表示，所以可以用另外一种不同的红黑树替代图 28-35a 中的树。由你来说明最后的结果是同样的，不过这个分裂过程中要做的工作更少些。(见练习 14。)

父结点为红结点的 4- 结点的分裂：情形 2。 图 28-34a 中的 4- 结点是其父结点的右孩子。28.37 如果它是左孩子，则红黑树表示将如图 28-36a 所示。这个图中的其他部分表明，要分裂 4- 结点，必须进行颜色翻转及右旋转。

如前所述，我们可以使用 3- 结点的另外一种红黑树表示，从而用另外一棵红黑树来替代图 28-36a，得到同样的最终结果。我们也将细节作为练习留给你来完成。(见练习 15。)

父结点为红结点的 4- 结点的分裂：情形 3 和情形 4。 现在考虑 4- 结点是其 3- 结点父结点中间孩子的情况。这次，3- 结点父结点产生的两种红黑树表示放在一起讨论。图 28-37a

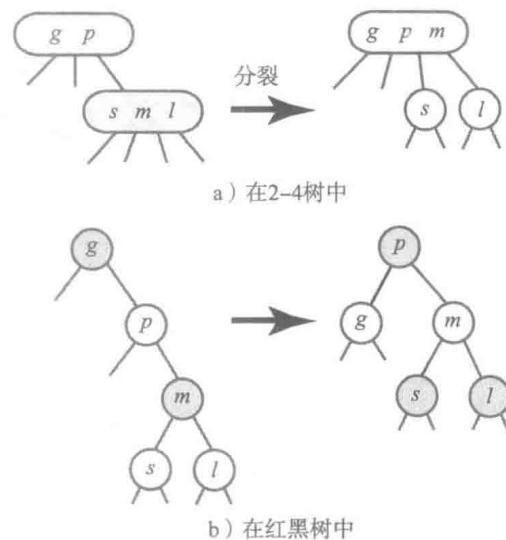


图 28-34 其父结点是 3- 结点的 4- 结点的分裂

表示的是红黑树的一种可能。图 28-37b 中颜色翻转后，我们解决了连续两个红结点问题，与图 28-31d 中的处理一样。右 - 左双旋转再加上颜色翻转，得到想要的结果，如图 28-37 中其他部分所示。

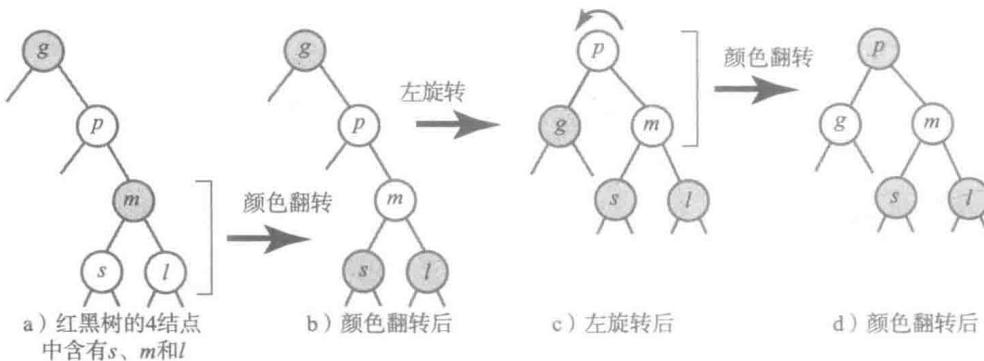


图 28-35 红黑树中父结点为红结点的 4- 结点的分裂：情形 1

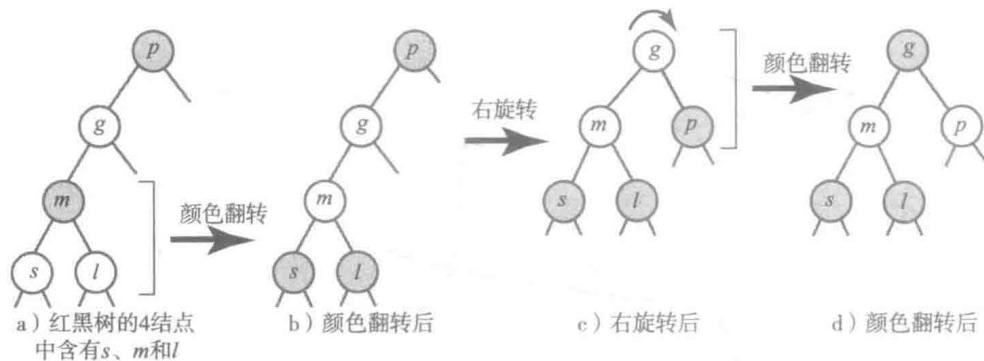


图 28-36 红黑树中父结点为红结点的 4- 结点的分裂：情形 2

图 28-38a 显示红黑树的第二种可能。图 28-38b 中颜色翻转解决了连续两个红结点问题，与图 28-32d 中的处理一样。还必须再进行左 - 右双旋转加上颜色翻转，如图 28-38 中其他部分所示。注意，图 28-38e 中的树与图 28-37e 中的树是一样的。

注：红黑 4- 结点的分裂

当分裂红黑 4- 结点时，其父结点的颜色决定了所需的操作。如果父结点是黑色的，则颜色翻转就足够了。但如果父结点是红色的，就必须进行颜色翻转、旋转及再次的颜色翻转。

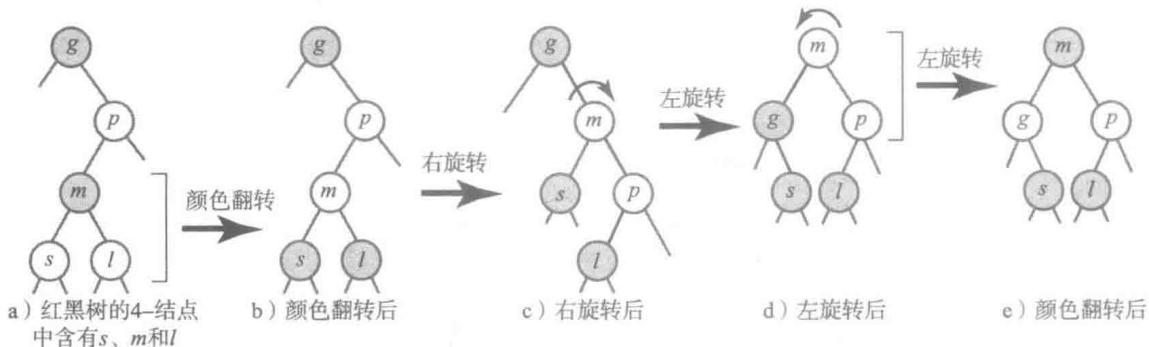


图 28-37 红黑树中父结点为红结点的 4- 结点的分裂：情形 3

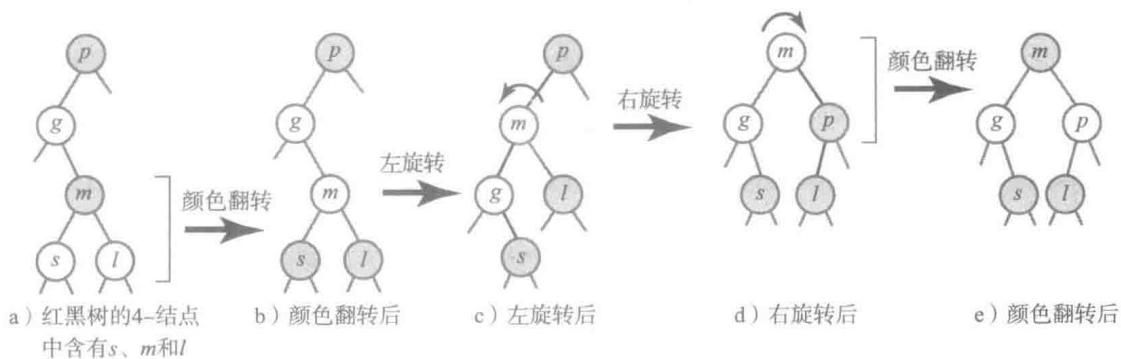


图 28-38 红黑树中父结点为红结点的 4- 结点的分裂：情形 4

Java 类库：类 TreeMap

包 `java.util` 中含有类 `TreeMap<K, V>`。这个类使用红黑树来实现同一包中接口 `SortedMap<K, V>` 中的方法。`SortedMap` 扩展了我们在第 20 章段 20.22 中描述的接口 `Map<K, V>`。^{28.39} 回忆一下，接口 `Map` 类似于我们为 ADT 字典使用的接口。`SortedMap` 规范说明了一个有序字典，其中查找键按升序排列。因为 `TreeMap` 使用了红黑树，所以像 `get`、`put`、`remove` 和 `containsKey` 这样的方法都是 $O(\log n)$ 的操作。

B 树

m 阶多路查找树 (multiway search tree of order m) ——有时称为 m 路查找树 (m -way search tree) ——是一棵每个结点最多有 m 个孩子的一般树。有 $k-1$ 个数据项和 k 个孩子的结点称为 k - 结点 (k -node)。 m 阶多路查找树可以含有 k - 结点， k 的取值范围为 $2 \sim m$ 。^{28.40}

二叉查找树是 2 阶多路查找树。你已经知道，不是所有的二叉查找树都是平衡的；同样，不是所有的多种查找树都是平衡的。但是，2-3 树和 2-4 树分别是 3 阶和 4 阶平衡的多路查找树。例如，我们要求 2-3 树的每个内部结点有 2 个或 3 个孩子，且所有的叶结点都在同一层，从而维护了树的平衡性。

m 阶 B 树 (B-tree of order m) 是 m 阶平衡多路查找树，使用下列附加特性来维护其平衡性：

- 根或者没有孩子，或者有 $2 \sim m$ 个孩子。
- 其他每个内部结点（非叶结点）都有 $\lceil m/2 \rceil$ 到 m 个孩子。
- 所有的叶结点都在同一层。

2-3 树和 2-4 树满足这些约束条件，所以它们是 B 树的例子。

到目前你见过的查找树都是在计算机主存中维护树中的数据。有时，我们或许要将数据保存在外存中，例如磁盘。只要我们能将数据读回内存，就可以使用之前的查找树。但当数据库太大不能完全放在内存中时该怎么办？通常我们使用一棵 B 树。^{28.41}

访问外存中的数据比访问主存中的数据要慢得多。当读取外部数据时，主要的开销用于在存储设备中找到数据。例如，磁盘中的数据顺序组织为块 (block)，其大小依赖于磁盘的物理特性。当从磁盘中读取数据时，整个块都被读出。找到块要比读取数据花更多的时间。如果每个块至少能包含一个结点的数据，则将很多的数据项放在一个结点中就能减少访问时间。虽然每个结点内可能要进行很多次比较，但这些操作的开销比访问外部数据要少得多。

因为每个结点中数据项数的增加降低了树的高度，故减少了你必须查找的结点个数，从而减少了磁盘的访问次数。高阶 B 树符合这些要求。为使 $m-1$ 个数据项能放到磁盘的同一个块中，可以选择 m 阶。

 注：虽然高阶 B 树因为每个结点内的比较次数增加，在用于内部数据库时往往适得其反，但用它来维护磁盘这样的外部存储器时很受人欢迎。

本章小结

- AVL 树是平衡的二叉查找树，当它不平衡时重排它的结点。如果添加一个结点导致了不平衡，则单旋转或双旋转就可以恢复树的平衡。
- 2- 结点是有两个孩子及一个数据项的结点。3- 结点有 3 个孩子和两个数据项。
- 2-3 树是含有 2- 结点和 3- 结点的平衡查找树。当向树中的添加操作可能导致叶结点中含有 3 个数据项时，叶结点将分裂为 2 个 2- 结点。这些结点含有 3 个数据项中的最小和最大项，并成为原叶结点之父结点的孩子结点。中间数据项提升到这个父结点中，有可能导致这个结点也要分裂。
- 2-3 树的缺点是，添加算法沿从根结点到叶结点的路径处理，然后结点分裂时又沿同一路经返回。
- 4- 结点有 4 个孩子和 3 个数据项。
- 2-4（或 2-3-4）树是含有 2- 结点、3- 结点和 4- 结点的一棵平衡查找树。向树中添加时，从根结点到叶结点的查找过程中，遇到的每个 4- 结点都分裂。所以，不需要沿路径返回到根结点。
- 红黑树是一棵逻辑上等价于 2-4 树的二叉查找树。概念上，红黑树比 2-4 树更难理解，但它的实现效率更高，因为它仅用到了 2- 结点。
- 在红黑树中的添加操作，使用颜色翻转及 AVL 树中用过的旋转，能维持树的平衡性和状态。
- k - 结点是含 k 个孩子和 $k-1$ 个数据项的结点。
- m 阶多路查找树是含 k - 结点的一般树， k 的取值范围为 $2 \sim m$ 。 m 阶 B 树是平衡的 m 阶多路查找树。为维持其平衡性，B 树需要每个内部结点都有确定个数的孩子，且所有的叶结点都在同一层中。
- 2-3 树是 3 阶 B 树；2-4 树是 4 阶 B 树。
- 当数据保存在如磁盘这样的外存时，B 树很有用。

练习

1. 实现段 28.5 给出的左 - 右双旋转算法。
2. 将 62 和 65 添加到图 28-27a 所示的 AVL 树中。
3. 将 62 和 65 添加到图 28-27b 所示的 2-3 树中。
4. 将 62 和 65 添加到图 28-27c 所示的 2-4 树中。
5. 将 62 和 65 添加到图 28-29 所示的红黑树中。
6. 图 28-27 和图 28-29 中的每棵树都含有相同的值。练习 2 ~ 练习 5 要求你将 62 和 65 添加到每棵树中。描述这些添加操作对每棵树的影响。

7. 与图 28-25b 所示的 2-4 树等价的红黑树是什么？
8. 当将值 10、20、30、40、50、60、70、80、90 和 100 添加到下列每棵初始为空的树中时，得到的结果树分别是什么？
- 一棵 AVL 树
 - 一棵 2-3 树
 - 一棵 2-4 树
 - 一棵红黑树
9. 将练习 8 所给的值添加到初始为空的二叉查找树中。将得到的树与练习 8 创建的各棵树进行比较。哪棵树的查找效率最高？
10. 针对下列每种树，画出含有 20 个值的最低可能的树。
- AVL 树
 - 2-3 树
 - 2-4 树
 - 红黑树
11. 针对下列每种树，画出含有 20 个值的最高可能的树。
- AVL 树
 - 2-3 树
 - 2-4 树
 - 红黑树
12. 使用伪代码，描述下列树的中序遍历。
- 2-3 树
 - 2-4 树
13. 考虑段 28.14 给出的用于 2-3 树的查找算法。虽然这个伪代码连同我们的图，都涉及 2- 结点和 3- 结点，但实现 2-3 树时常常仅使用 3- 结点。可以首先考虑结点中的较小对象。然后，如果发现 `null` 替代了较大对象时，将这个结点看作 2- 结点。修改这个查找算法的伪代码，以表达出这些实现细节。
14. 图 28-34a 显示的是 2-4 树中的一个 4- 结点，它是其含 g 和 p 数据项的 3- 结点父结点的右孩子。当将这些结点转换为红黑结点表示时，令 p 为 g 的父结点。修改图 28-35，说明要得到所需的红黑树，只需要进行颜色翻转。
15. 这次假定 4- 结点是左孩子，重做练习 14。
16. 对图 28-39 中的每棵树用颜色标记结点，使其为一棵红黑树。

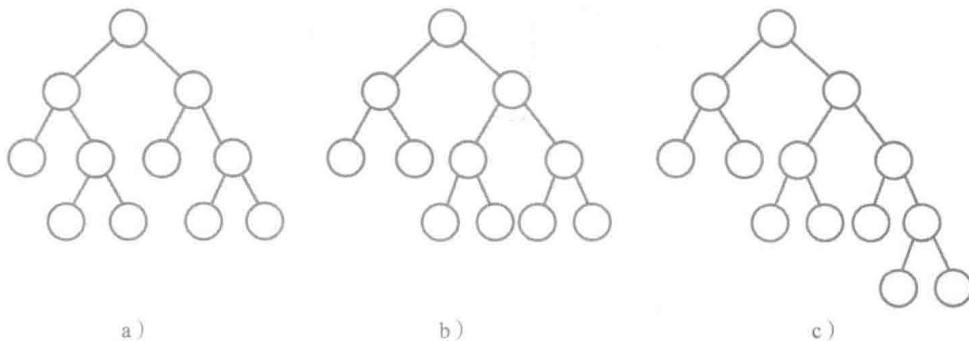


图 28-39 用于练习 16 的三棵二叉树

17. 本章学习的哪种树能用来实现优先队列？回忆我们在第 7 章讨论过优先队列。
18. 用红黑树或 AVL 树实现优先队列的 `add` 和 `remove` 方法的效率如何？
19. 考虑高度为 3 的 1000 阶 B 树。这棵树中能保存的值最少是多少？最多是多少？将你的结果推广到高度为 h 的 1000 阶 B 树。

项目

- 使用从二叉查找树中删除项的方式，从 AVL 树中删除一项。只是，从树中删除相应的结点后可能会出现不平衡，你必须执行单旋转或双旋转来修正它。开发一个算法从 AVL 树中删除一个结点。
- 实现 AVL 树的类。
- 考虑段 28.11 给出的用于 AVL 树的方法 `rebalance` 的实现。这个方法的性能依赖于旋转的开销和方法 `getHeightDifference` 的开销。

- a. 假定树的高度为 h , `nodeN` 的高度为 k 。使用大 O 表示法, 下面每个任务的开销是多少?
- 旋转
 - 执行方法 `getHeightDifference`
 - 执行方法 `rebalance`
- b. 假定结点通过方法 `add` 添加在树的底部。`rebalance` 将被调用多少次? 给出添加一个结点操作的大 O 表示。
4. 设计并实现可以用来实现 AVL 树的结点类。可以从 `BinaryNode` 类派生这个类。作为子树根的每个结点, 应该含有这棵子树的高度。使用你的新结点类实现 AVL 树的类。
5. 设计可以用来实现 2-4 树的结点类。一个类够吗? 还是需要多个类?
6. 练习 13 要求你修改段 28.14 给出的用于 2-3 树的查找算法。实现你修改后的伪代码。
7. 实现 2-4 树的类, 其中仅允许添加和获取操作。
8. 实现红黑树的类, 其中仅允许添加和获取操作。
9. 实现将项保存在红黑树中的集合类, 红黑树是前一个项目中定义的类的实例。你实现的类中可以忽略方法 `remove` 和 `clear`。回忆第 1 章中将集合定义为一个不允许有重复值的包。
10. 设计并完成将普通二叉查找树的高度与 AVL 树或红黑树的高度进行比较的实验。你应先完成项目 2 或项目 8。
11. 设计 m 阶 B 树的结点类 `BTreeNode`。考虑允许下列操作。
- 获得结点内的具体值
 - 将值插入结点中, 同时维护与子树的链接 (记住, B 树是一棵查找树)
 - 获得结点中值的个数
 - 替换结点中的值, 如果查找树仍保持
 - 替换一棵子树
 - 将结点分裂为两个结点, 每个结点含有一半的值
 - 给出将新值添加到用你自己定义的 `BTreeNode` 实现的 B 树中的算法。
12. 使用本章的一种平衡查找树实现优先队列。

先修章节：第 5 章、第 7 章、第 24 章

目标

学习完本章后，应该能够

- 描述图的特征，包括顶点、边和路径
- 给出图的示例，包括无向图、有向图、无权图和带权图
- 给出有向图及无向图中邻接顶点及不是邻接点的示例
- 给出路径、简单路径、回路及简单回路示例
- 给出连通图、不连通图及完全图示例
- 在给定图上执行深度优先遍历和广度优先遍历
- 列出有向无环图顶点的拓扑序
- 检测图中给定两顶点间是否存在路径
- 寻找一个顶点到另一顶点间的最少边数的路径
- 在带权图中，寻找一个顶点到另一个顶点间的最小代价路径
- 描述用于 ADT 图的操作

新闻媒体常使用线图、饼图或柱状图帮我们将某些统计数据形象化。但这些常见的图不是本章要研究的这类图的示例。计算机科学家和数学家使用的图包括了你在第 24 章中见过的树。事实上，树是一类特殊的图。这些图表示数据元素之间的关系。本章将介绍讨论图时用到的术语、图上的操作及一些典型应用。

一些示例及术语

虽然你过去画过的图，可能不是我们要在这里讨论的图，但本节的示例还是很常见的。不过你可能从没把它们称为图。

公路地图

图 29-1 所示为马萨诸塞州科德角公路地图的一部分。小圆圈表示城镇，连接它们的线表示公路。公路地图是一幅图。图中，圆圈称为顶点 (vertex)，或结点 (node)，而线称为边 (edge)。图 (graph) 是不同顶点和不同边的集合。子图 (subgraph) 是图的一部分，自身也是一个图，正如图 29-1 所示的公路地图实际上就是一张更大地图的一部分。

因为可以沿图 29-1 中的公路双向移

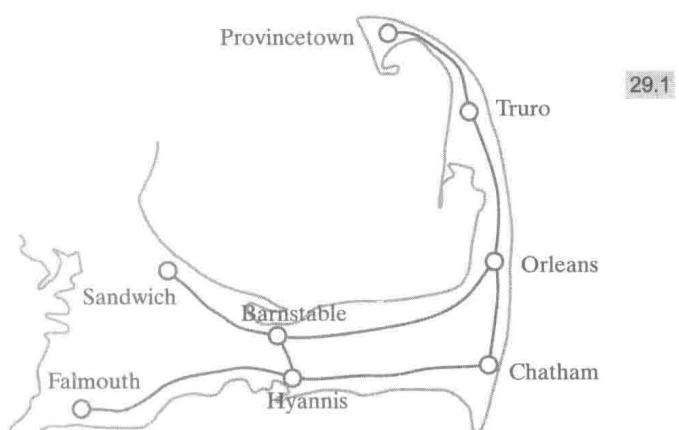


图 29-1 公路地图的一部分

动，所以相应的图及其边称为无向的（undirected）。但城市常有单行路。图 29-2 中在城市街道地图中的交叉路口都有一个顶点。

每条边有一个方向，所以称为有向边（directed edge）。带有有向边的图称为有向图（directed graph，简写为 digraph）。可以将每条无向边替换为一对方向相对的有向边，从而将无向图转为有向图。

29.2 路径。图中两个顶点间的路径（path）是边的序列。有向图中的路径必须考虑边的方向，故称为有向路径（directed path）。路径长度（length）是路径所含的边数。如果路径经过的顶点不重复，则称为简单路径（simple path）。图 29-1 含有从 Provincetown 到 Orleans 的长度为 2 的简单路径。

回路（cycle）是开始顶点及结束顶点均为同一顶点的路径。经过其他顶点仅一次的回路称为简单回路（simple cycle）。图 29-1 中，回路 Chatham-Hyannis-Barnstable-Orleans-Chatham 是简单回路。没有回路的图称为无环图（acyclic）。

使用公路地图或是街道地图查看如何从 A 点到达 B 点。选择的这些点之间的路径通常是一条简单路径。这样做可以避免折回去走回头路或是在环上转圈。但去观赏秋叶的人们，会走出一条回路，起点和终点都是家的所在地。

29.3 权。你可能很高兴从一个地方到另一个地方，但会常常选择不同的路径。例如可能选择最短的、最快的或是最便宜的路径。这样用到了带权图（weighted graph），其边上有一个值。这些值称为权（weight）或代价（cost）。例如，图 29-1 中的公路图作为带权图显示在图 29-3 中。这种方式下，每个权表示两个城镇之间距离的英里数。你可能会用到表示驾驶时间或是乘出租旅行费用的其他类型的权。

带权图中的路径也有一个权或代价，它是边权值的和。例如，图 29-3 中从 Provincetown 到 Orleans 的路径权值为 27。

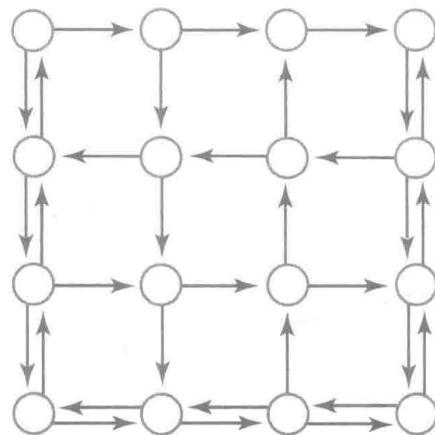


图 29-2 表示城市部分街道地图的有向图

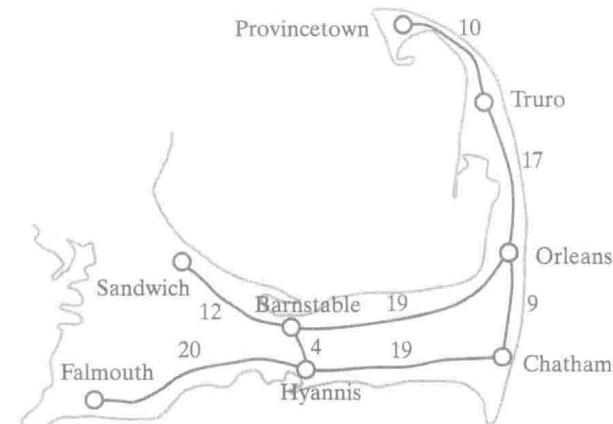


图 29-3 带权图

学习问题 1 考虑图 29-3 中的图。

- a. 从 Provincetown 开始，经过 Truro 和 Orleans，到 Chatham 结束的路径长度是多少？
- b. 刚刚描述的路径权值是多少？
- c. 考虑从 Truro 到 Sandwich 不带回路的所有路径。哪条路径具有最短长度？
- d. 上一问中提到的所有路径中，哪条路径具有最小权值？

以从这里到达那里。每对不同顶点间都有路径可达的图称为连通的 (connected)。完全图 (complete graph) 还不止如此；它在每对不同顶点间都有一条边。图 29-4 中提供了无向图的几个示例，分别是连通图、完全图及非连通图 (disconnected)，即不连通的。注意，图 29-4a 中的简单路径和图 29-4c 中的简单回路。

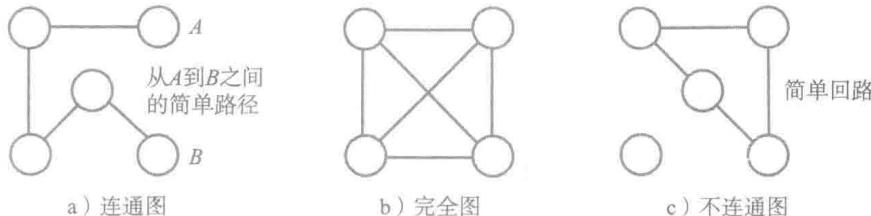


图 29-4 无向图

邻接顶点。如果无向图中的两个顶点由边相连，则它们称为邻接的 (adjacent)。29.5图 29-3 中，Orleans 和 Chatham 是邻接的，但 Orleans 和 Sandwich 不是邻接的。邻接顶点称为邻居 (neighbor)。在有向图中，如果存在一条起始于 j 终止于 i 的有向边，则称顶点 i 与顶点 j 邻接。在图 29-5 中，顶点 A 与顶点 B 邻接，但顶点 B 不与顶点 A 邻接。即顶点 A 是顶点 B 的邻居，但反过来是不成立的。

为了方便起见，我们用顶点标号外加圆圈来表示顶点，如图 29-5 所示。但有时，顶点标号会出现在圆圈旁边，如图 29-3 所示。

边数。如果有向图含有 n 个顶点，那么它能含有多少条边？如果图是完全图，则每个顶点都是其余所有顶点的邻居。所以每个顶点都是 $n-1$ 条有向边的终止点。因此，有 $n \times (n-1)$ 条边。无向完全图中的边数是这个数字的一半。例如，图 29-4b 中的图有 4 个顶点和 $(4 \times 3)/2$ 即 6 条边。要使这个图成为有向图且是完全图，每条边都要用两条有向边来替换，得到有 12 条边的图。29.6

图 29-5 顶点 A 与顶点 B 邻接，但顶点 B 不与顶点 A 邻接

注：如果图中有 n 个顶点，则它最多有

- $n \times (n-1)$ 条边，如果图是有向的
- $n \times (n-1)/2$ 条边，如果图是无向的

如果图中含有相对较少的边，则图是稀疏图 (sparse)。如果它含有很多边，则图是密集图 (dense)。这些术语没有精确定义，我们说稀疏图有 $O(n)$ 条边，而密集图有 $O(n^2)$ 条边。图 29-1 中的图有 8 个顶点和 8 条边。它是稀疏图。29.7

注：一般的图是稀疏图。

航空公司的航线

表示航空公司飞机飞行航线的图，类似于表示公路地图的图。但它们也有差别，因为不是每个城市都有机场，不是每个航空公司在每个机场都能到达或起飞。例如，图 29-6 中的图表示的是美国东海岸一家小型航空公司的航线。图是无向的，由两个连通子图组成。但整个图是不连通的。

注意，你可以从 Boston 飞往 Provincetown，但不能从 Boston 飞往 Key West。可以编写一个算法来查看给定城市之间是否有航班。

 注：图 29-6 是由两个不同子图组成的一个图。虽然每个子图都是连通的，但整个图不是连通的。

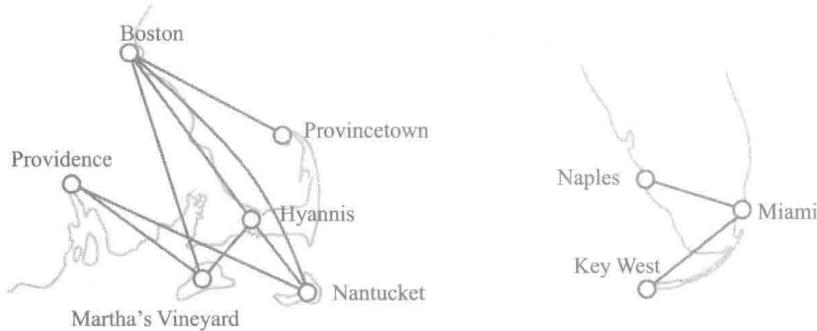


图 29-6 航空公司的航线

迷宫

29.8 可以在入口、出口、路径上的每个转弯处及每个死胡同处都放置一个顶点，从而将迷宫表示为一个图，如图 29-7b 所示。这个图是连通的，很像是图 29-1 所示的公路地图。对这样的图，我们可以找到任意两个顶点间的一条路径，本章后面会讨论。

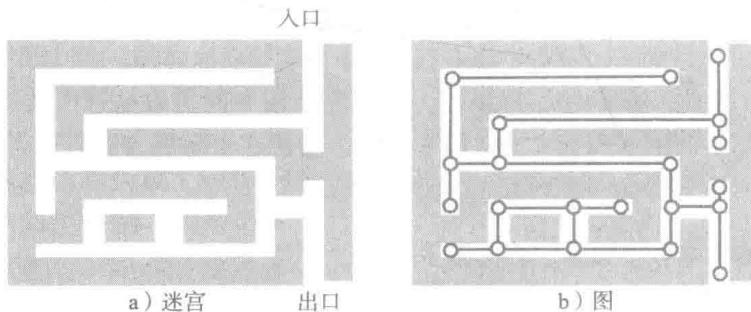


图 29-7 迷宫及将它表示为一个图

先修课程

29.9 作为大学生，必须学习本专业的一系列课程。每门课程都有一些必须先学习的先修课程。以什么次序学习所需课程能满足先修条件呢？

要回答这个问题，先创建一个表示课程及先修关系的有向图。图 29-8 是这种图的一个示例。每个顶点表示一门课程，每条有向边始点的课程是另一门课程的先修课。例如，在学习 cs10 之前必须先学习 cs1、cs2、cs4、cs7、cs9 和 cs5。

这个图没有回路。在有向无环图中，当从 a 到 b 之间存在一条有向边时，让顶点 a 排在顶点 b 的前面，从而可以排列图

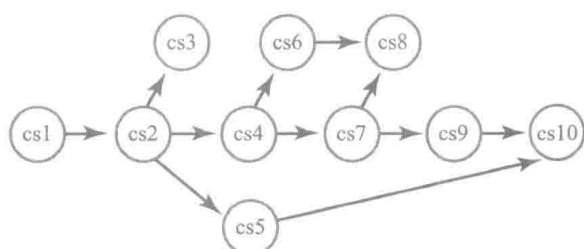


图 29-8 将选择课程的必要条件表示为有向无环图

中的顶点。这样排列的顶点次序称为拓扑序 (topological order)。本章后面将讨论如何找到这个次序，因此，这个次序就是完成课程要求的次序。

树

ADT 树是使用父 – 子关系将结点组织为层次关系的一种图。特殊的结点根是树中所有其他结点的祖先。但不是所有的图都有层次结构，所以不是所有的图都是树。29.10



注：所有的树都是图，但不是所有的图都是树。树是无环的连通图。



学习问题 2 一所典型房子中的哪些物理系统能表示为图？

学习问题 3 图 29-1 中的图是连通的吗？是完全图吗？

学习问题 4 图 29-8 中的图是树吗？请解释。

学习问题 5 对于图 29-8 中的图，

- a. cs1 与 cs2 邻接吗?
- b. cs2 与 cs1 邻接吗?
- c. cs1 与 cs4 邻接吗?
- d. cs4 与 cs1 邻接吗?

遍历

在前面几章学过，可以在树中查找包含某个值的结点。但图的应用集中在顶点的连通上，而不是顶点的内容上。这些应用常常基于图顶点的遍历。29.11

在第 24 章中，讨论了访问树结点的几种次序。前序、中序和后序遍历是深度优先遍历的例子。这类遍历沿树中一条越来越深的路径访问，直到到达叶结点，如图 29-9a 所示。更一般地，图的深度优先遍历沿图中一条越来越深的路径访问，然后再沿其他路径访问。访问一个顶点后，遍历访问该顶点的邻居，邻居的邻居，依此类推。

树的层序遍历是广度优先遍历的例子。它沿一条检查整个层的路径访问，然后移到下一层，如图 29-9b 所示。在图中，广度优先遍历访问一个结点的所有邻居，然后访问邻居的邻居。

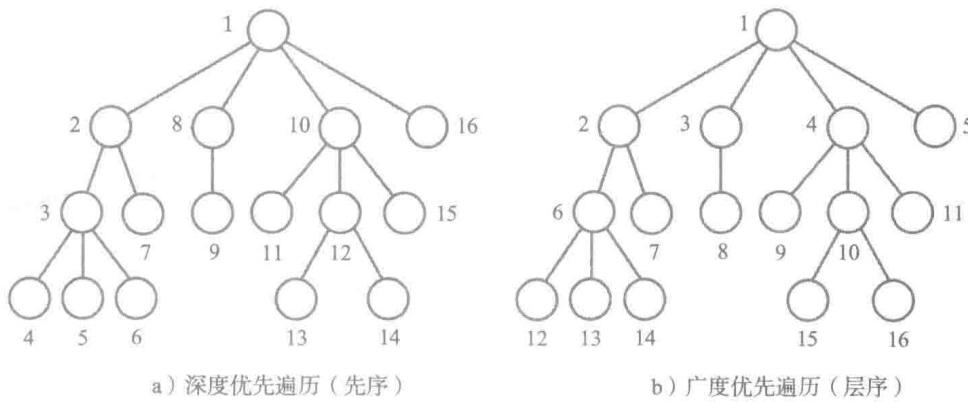


图 29-9 两种遍历的访问次序

注：访问树或图中的结点，是遍历过程中要执行的动作。在树中，“访问结点”意味着“处理结点中的数据”。在图中，“访问结点”意味着“将结点标记为已访问过”。

树的遍历由根结点开始访问树中的所有结点。但是图的遍历从任何顶点——称为起始顶点 (origin vertex) ——开始，仅访问能到达的顶点。仅当图是连通图时，这样的遍历才能访问所有的顶点。

广度优先遍历

29.12 给定一个起始顶点，广度优先遍历访问起始顶点，然后访问起始顶点的各邻居。再然后对这些邻居中的每一个，访问它的邻居。遍历使用队列保存访问过的顶点。当从队列中删除一个顶点时，访问它并将该顶点未被访问的各邻居入队列。则遍历次序就是顶点入队列的次序。可以将这个遍历次序保存在第二个队列中。

下面算法从非空图的一个给定顶点开始执行广度优先遍历。

```
Algorithm getBreadthFirstTraversal(originVertex)
traversalOrder=用来保存得到的遍历次序的新队列
vertexQueue=用来保存要访问的结点的新队列
标记originVertex为已访问
traversalOrder.enqueue(originVertex)
vertexQueue.enqueue(originVertex)
while (!vertexQueue.isEmpty())
{
    frontVertex = vertexQueue.dequeue()
    while(frontVertex有一个邻居)
    {
        nextNeighbor=frontVertex的下一个邻居
        if (nextNeighbor未被访问)
        {
            标记nextNeighbor为已访问
            traversalOrder.enqueue(nextNeighbor)
            vertexQueue.enqueue(nextNeighbor)
        }
    }
}
return traversalOrder
```

图 29-10 跟踪了该算法用于有向图的过程。

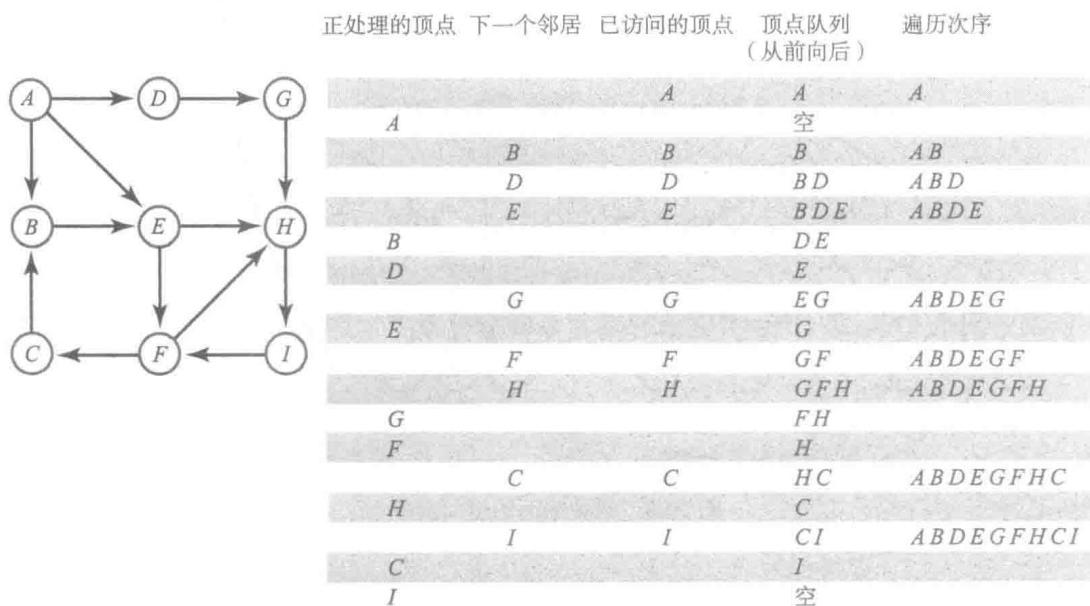


图 29-10 从顶点 A 开始广度优先遍历有向图的过程

**注：广度优先遍历**

广度优先遍历访问一个顶点，然后在推进之前访问顶点的每个邻居。访问邻居的次序可以任意，可以依赖于图的实现。



学习问题 6 广度优先遍历图 29-10 所示的图，从顶点 E 开始，按字母序访问邻居，得到的顶点次序是什么？

深度优先遍历

给定起始顶点，深度优先遍历访问起始顶点，然后访问起始顶点的一个邻居，再然后访问邻居的邻居。持续这个过程直到发现没有未访问的邻居时为止。回退一个顶点，再检查它的另一个邻居。这个遍历有递归的感觉，因为从起始顶点的遍历导致从起始顶点的邻居的遍历。因此你不会讶异于给出这个遍历的递归描述时用到了栈。

29.13

初始时将起始顶点入栈。当栈顶顶点有未被访问的邻居时，我们访问它并将它的邻居入栈。如果不存在这样的邻居，则出栈。遍历次序是顶点入栈的次序。可以将这个遍历次序保存在一个队列中。

下面算法从非空图的一个给定顶点开始执行深度优先遍历。

```
Algorithm getDepthFirstTraversal(originVertex)
traversalOrder=用来保存得到的遍历次序的新队列
vertexStack=用来保存要访问的结点的新栈
标记originVertex为已访问
traversalOrder.enqueue(originVertex)
vertexStack.push(originVertex)

while (!vertexStack.isEmpty())
{
    topVertex = vertexStack.peek()
    if (topVertex有一个未访问的邻居)
    {
        nextNeighbor=topVertex的下一个未访问的邻居
        标记nextNeighbor为已访问
        traversalOrder.enqueue(nextNeighbor)
        vertexStack.push(nextNeighbor)
    }
    else // All neighbors are visited
        vertexStack.pop()
}
return traversalOrder
```

图 29-11 跟踪了该算法用于图 29-10 所示同一有向图的过程。

**注：深度优先遍历**

深度优先遍历访问一个顶点，然后访问顶点的一个邻居，邻居的邻居，依此类推，从起始顶点开始尽可能地向远处推进。然后回退一个顶点并检查另一个邻居。访问邻居的次序可以任意，可以依赖于图的实现。



学习问题 7 深度优先遍历图 29-11 所示的图，从顶点 E 开始，按字母序访问邻居，得到的顶点次序是什么？

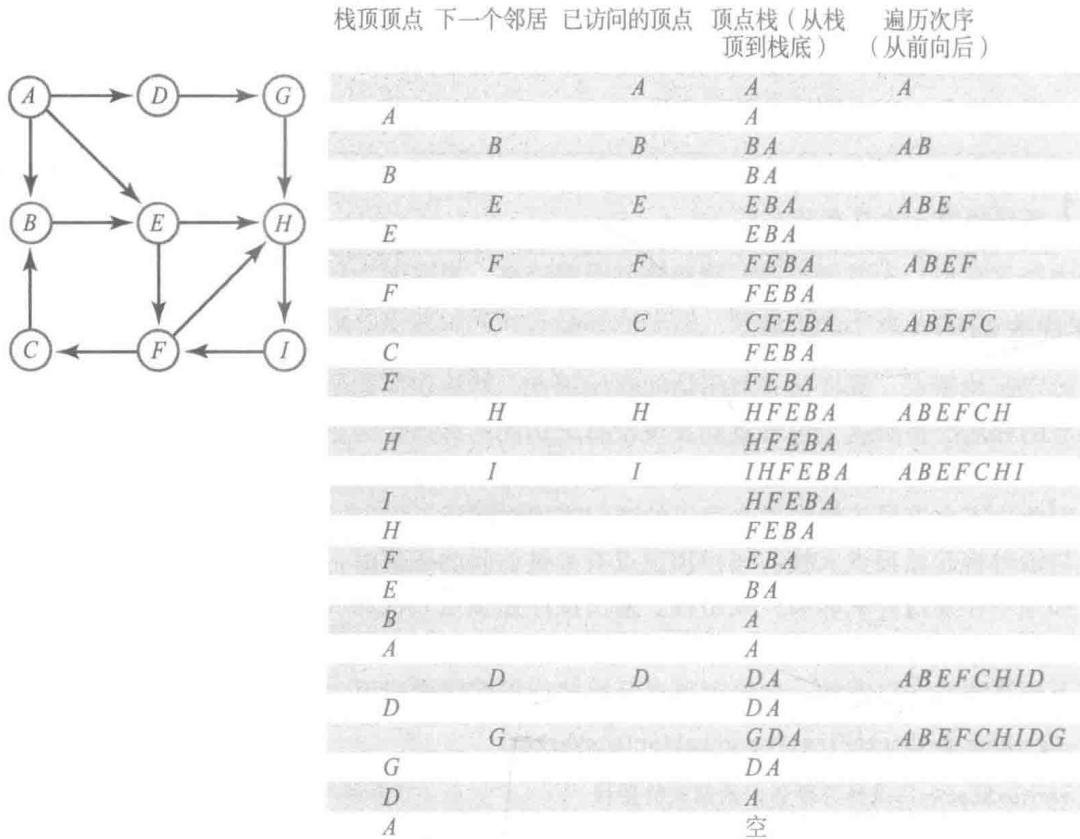


图 29-11 从顶点 A 开始深度优先遍历有向图的过程

拓扑序

29.14

图 29-8 是表示一组计算机科学课程的先修结构图。这个图是有向无环图。回忆一下，你可以将这样一个图的顶点按拓扑序排列。



注：在有向无环图顶点的拓扑序中，当从顶点 a 到 b 有有向边时，顶点 a 排在顶点 b 的前面。

图中的顶点可以有几种不同的拓扑序。例如，图 29-8 所示图的一种拓扑序是 cs1、cs2、cs5、cs4、cs7、cs9、cs10、cs6、cs8、cs3。即如果以这个次序完成课程，则可以满足所有先决条件。假定可以移动图中的顶点，让它们按这种次序排成一排，相应地按需拉伸边。得到的图会像图 29-12a 所示。每条边指向边起始结点之后的结点。对每个有向图，如果图中没有回路，则至少能找到一种这样的排列。图 29-12 中还显示了图 29-8 的另外两种拓扑序。与本例一样，任何一种拓扑序通常都能解决给定的问题。

有环图不可能得到拓扑序。如果顶点 a 和 b 在一个回路上，则从 a 到 b 和从 b 到 a 都存在路径。我们选择 a 、 b 的任何次序，都会与这些路径中的一条相矛盾。例如，图 29-13 所示的图中含有一个回路。在学习 cs30 之前必须完成 cs15 和 cs20。但你在学习 cs20 之前必须完成 cs30。这个循环逻辑是回路造成的，且导致了一种不可能的情形。

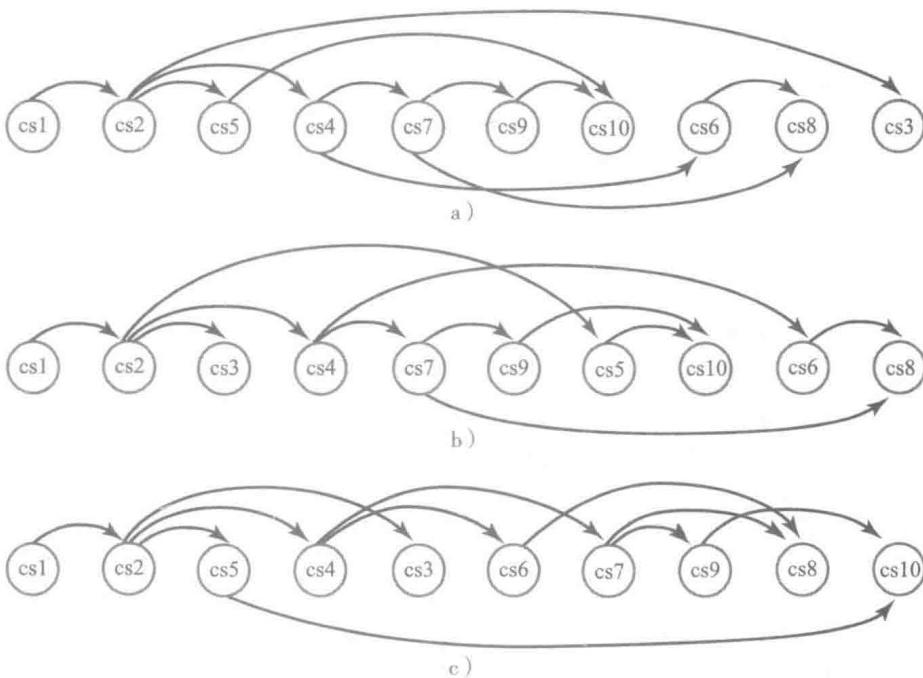


图 29-12 图 29-8 所示图的三种拓扑序



图 29-13 因为是一个带回路的有向图，故 3 门课程的先修结构是不可能的



学习问题 8 图 29-8 所示图中顶点的另一个拓扑序是什么？

29.15

发现图中顶点拓扑序的过程称为 **拓扑排序** (topological sort)。有可能有几个算法。我们从找到没有后继的顶点——即没有邻接顶点——开始拓扑排序过程。找到这个顶点是可能的，因为图中没有回路。将这个顶点标记为已访问并入栈。继续查找另一个未访问过的而其已有的邻居均已访问过的顶点 u 。标记 u 为已访问并将其入栈。继续这个过程，直到访问了所有顶点为止。此时，栈中从栈顶开始保存的即是顶点的拓扑序。

下列算法描述了这个拓扑排序过程。

```

Algorithm getTopologicalOrder()
vertexStack=用来保存被访问顶点的新栈
numberOfVertices=图中顶点个数
for (counter=1 到 numberOfVertices)
{
    nextVertex=其已有的邻居都已被访问的未访问顶点
    标记 nextVertex 为已访问
    vertexStack.push(nextVertex)
}
return vertexStack

```

图 29-14 跟踪了这个算法用于图 29-8 所示图的过程。算法循环的每次迭代中，当访问下一个未访问顶点 (nextVertex) 时在图中用阴影标记它。拓扑序与标记阴影的次序相反。本例得到的拓扑序是图 29-12a 中的一种。

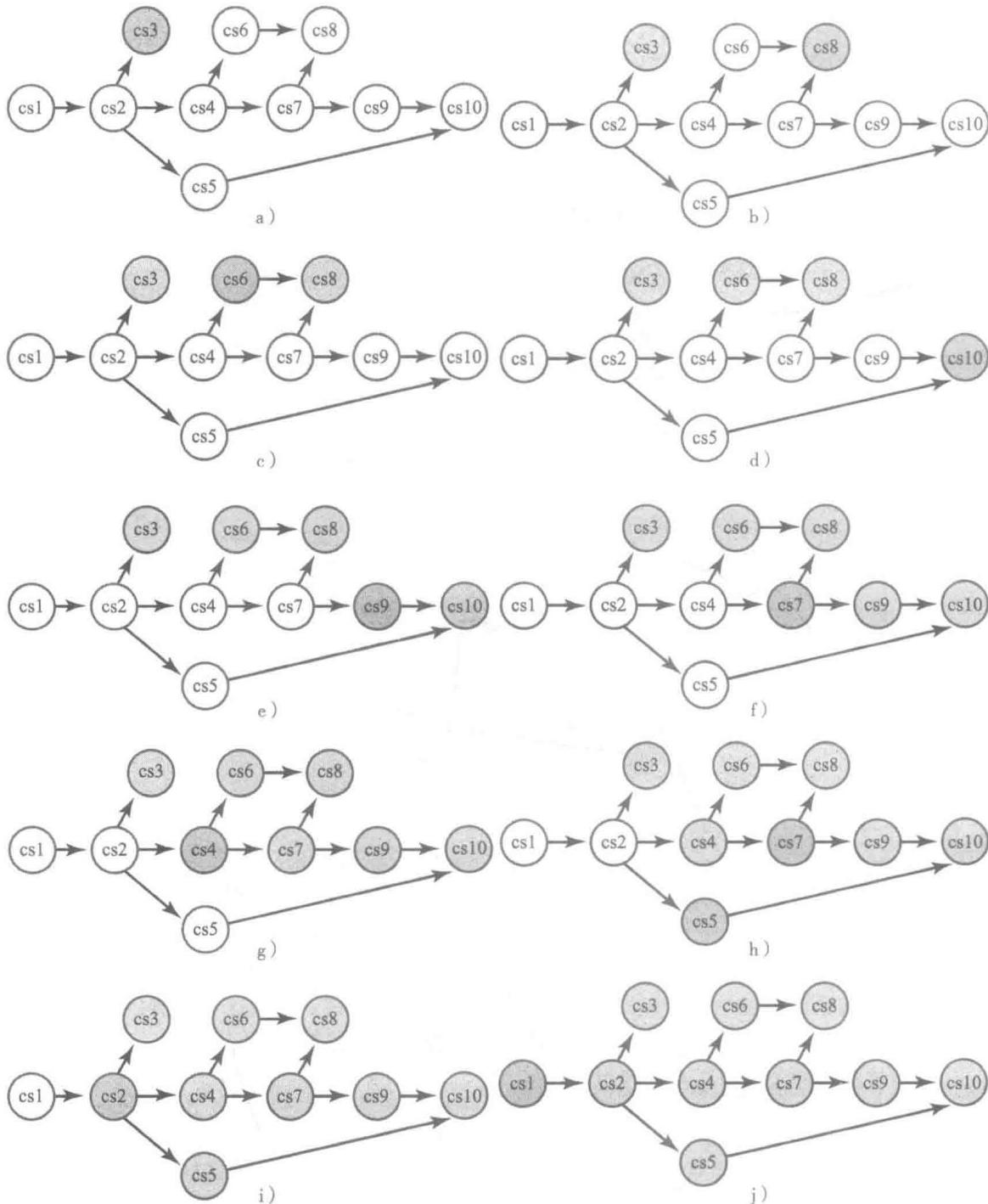


图 29-14 寻找图 29-8 所示图的一个拓扑序

路径

了解两个给定城市间是否有某一航空公司飞行对普通旅客是很重要的。用一个图——如图 29-6 这样的图——来表示航空公司的航线，并测试从顶点 a 到顶点 b 之间是否存在路径，就可以获得这个信息。如果有路径存在，则可以找出这条路径。如果不是所有路径都一样，则可以找一条最短或最便宜的路径。

寻找路径

就眼前来说，我们满足于寻找任何一条路径，不一定是最佳路径。深度优先遍历——段 29.13 中所讨论的——沿着一条路径访问尽可能多的顶点。可以简单地修改这个遍历以便能找到两个顶点间的一条路径。从起始顶点开始。每次访问另一个顶点时，看看那个顶点是否是所要到达的目标顶点。如果是，则任务完成了，得到的栈中包含这条路径。否则继续遍历过程，直到或者成功，或者遍历结束。将这个算法留作练习。

29.16

无权图中的最短路径

示例。图中同一对顶点间可能有几条不同的路径。在无权图中，可以找到具有最短长度的路径，即路径上含有的边数最少。例如，考虑图 29-15a 所示的无权图。假定我们想知道顶点 A 到顶点 H 间的最短路径。通过对图的观察可以发现这两个顶点间有几条简单路径——列在图 29-15b 中。从 A 到 E 再到 H 的路径长度是 2，所以是最短的。

29.17

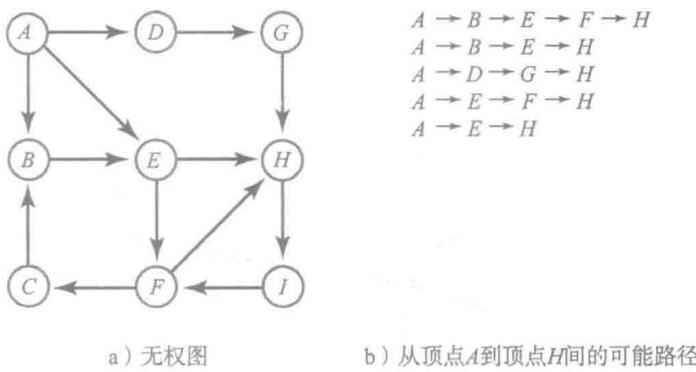


图 29-15 一个无权图及其从顶点 A 到顶点 H 间的可能路径

开发算法。寻找无权图中两个给定顶点间的最短路径的算法，可以基于广度优先遍历。回忆一下，这个遍历访问起始顶点，然后访问起始顶点的邻居，然后是这些邻居的邻居，以此类推。访问时将每个顶点放入队列。

29.18

要找到最短路径，可以如下增加广度优先遍历算法的处理步骤。当访问顶点 v 时，将其标记为已访问，记录下到达 v 之前刚刚离开的顶点 p 。即在图中， p 在 v 的前面。还记下遍历到 v 的路径长度。这个长度比到达顶点 p 的路径长度长 1。将到达 v 的路径长度和指向顶点 p 的引用保存在顶点 v 中。每个顶点都包含一个标号、到达该顶点的路径长度，及该路径上在它前面的那个顶点，如图 29-16 所示。

虽然一个顶点中还含有其他的数据域，但在图中我们省略了它们。遍历结束后，将使用顶点中的这些数据来构造最短路径。

我们跳到算法部分。对图 29-15a 所示的图，算法从顶点 A 遍历到顶点 H 后图的状态，展示在图 29-17a 中。

现在，通过检查目标顶点 H，我们发现从 A 到 H 的最短路径长度是 2。还发现这条最短路径上 H 的前驱顶点是顶点 E。从顶点 E，能够发现它的前驱顶点是顶点 A。所以要找的从顶点 A 到顶点 H 的最短路径是 $A \rightarrow E \rightarrow H$ 。算法找到了最短路径，当然通过检查这个简单的图，我们已经知道找到的答案是正确的。



图 29-16 顶点中的数据

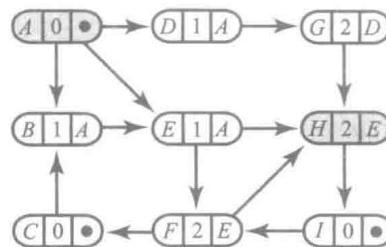


图 29-17 最短路径算法对图 29-15a 从顶点 A 遍历到顶点 H 后的图

注：在无权图中，两个给定顶点间的最短路径具有最短长度，即它有最少的边。找到这条路径的算法基于广度优先遍历。如果几条路径有同样的最短长度，则算法将找到其中的一条。

29.19 算法。下列算法找到无权图中顶点 `originVertex` 和 `endVertex` 之间的最短路径。与段 29.12 中广度优先遍历一样，算法使用队列来保存遍历过的顶点。然后使用所给的初始为空的栈 `path` 来构造最短路径。

```

Algorithm getShortestPath(originVertex, endVertex, path)
done = false
vertexQueue=用来保存要访问的结点的新队列
标记originVertex为已访问
vertexQueue.enqueue(originVertex)
while (!done && !vertexQueue.isEmpty())
{
    frontVertex = vertexQueue.dequeue()
    while (!done && frontVertex有一个邻居)
    {
        nextNeighbor=frontVertex的下一个邻居
        if (nextNeighbor未访问)
        {
            标记nextNeighbor为已访问
            将到nextNeighbor的路径长度赋为到frontVertex的路径长度加1
            nextNeighbor的前驱顶点赋为frontVertex
            vertexQueue.enqueue(nextNeighbor)
        }
        if (nextNeighbor等于endVertex)
            done = true
    }
}

// Traversal ends; construct shortest path
pathLength=到 endVertex的路径长度
path.push(endVertex)

vertex = endVertex
while (vertex 有前驱顶点)
{
    vertex=vertex的前驱顶点
    path.push(vertex)
}
return pathLength

```

当算法结束时，栈 `path` 中包含了最短路径上的顶点，初始顶点在栈顶。返回值是最短路径的长度。

跟踪算法。图 29-18 跟踪了算法应用于图 29-15a 中无权图时得到如图 29-17 所示的路径信息的步骤。将初始顶点——顶点 A——添加到队列后，访问初始顶点的 3 个邻居——B、

D 和 *E*——将它们入队列。

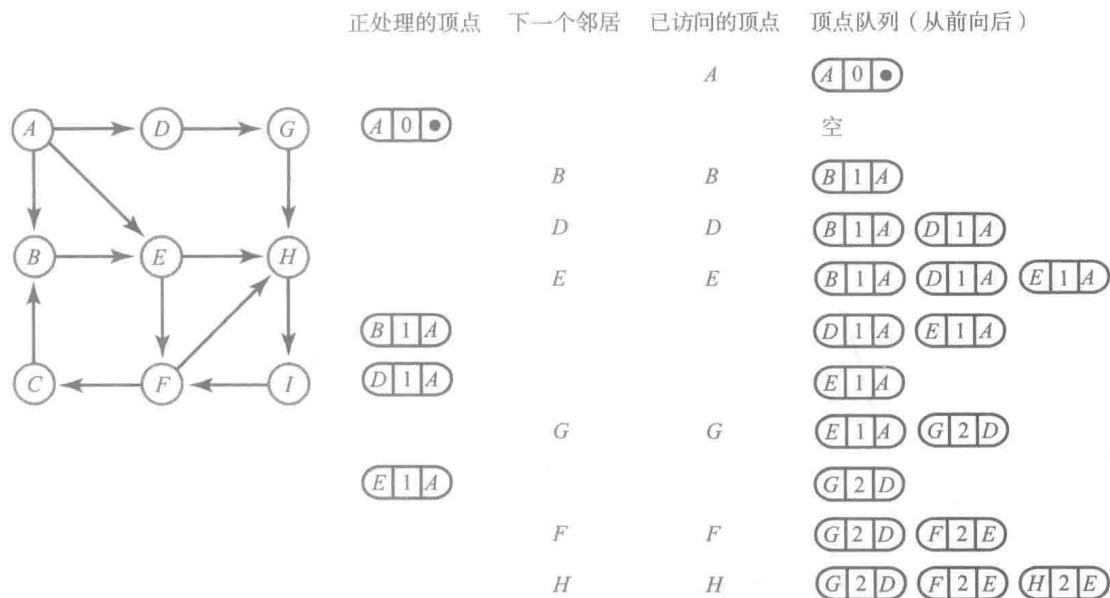


图 29-18 跟踪算法为得到无权图中从顶点 *A* 到顶点 *H* 间最短路径的遍历过程

从 *A* 到这些邻居的每条路径长度都是 1。因为顶点 *A* 没有其他的邻居了，所以从队列中删除顶点 *B*。这个顶点的邻居有 *E*，但 *E* 已经被访问过了。这意味着我们可以从 *A* 到达 *E* 不需要先通过 *B*。即 *B* 不在从 *A* 经过 *E* 的任何最短路径上。实际上，路径 *A* → *B* → *E* 比路径 *A* → *E* 要长。我们不知道最终的路径是否含有 *E*，但如果经过 *E*，则它不需要经过 *B*。

现在算法从队列中删除顶点 *D*。它的邻居 *G* 是未访问过的，所以将 *G* 的路径长度置为 2，它的前驱顶点置为 *D*。然后将 *G* 入队列。继续这个过程，最终会遇到目标顶点 *H*。当更新 *H* 的数据域后，外层循环结束。然后从 *H* 回退以便构造路径，如之前在段 29.18 中所论述的。



学习问题 9 继续图 29-18 所示的跟踪过程，找到从顶点 *A* 到顶点 *C* 的最短路径。

带权图中的最短路径

示例。在带权图中，最短路径不一定具有最少的边数。而是具有最小边权值总和的路径。在图 29-15a 中增加了权值，得到的带权图如图 29-19a 所示。从顶点 *A* 到顶点 *H* 的所有可能路径与图 29-15b 中看到的相同。只是这次我们在图 29-19b 中还列出了每条路径的权——即边权值之和。

可以看到，最短路径的权值是 8，即最短路径是 *A* → *D* → *G* → *H*。当权值是距离时，术语“最短”是合适的。当权值表示代价时，可以把这条路径看作“最便宜”的路径。

开发算法。找到带权图中两个给定顶点间最短或最便宜路径的算法，基于广度优先遍历。类似于我们为无权图开发的算法。在那个算法中，我们标注出通向当前顶点的路径中的边数。这里，我们计算出通向一个顶点的路径中边权值之和。另外，还必须记录可能的最短路径。之前我们使用队列来排序顶点，而本算法使用优先队列。

29.21

29.22

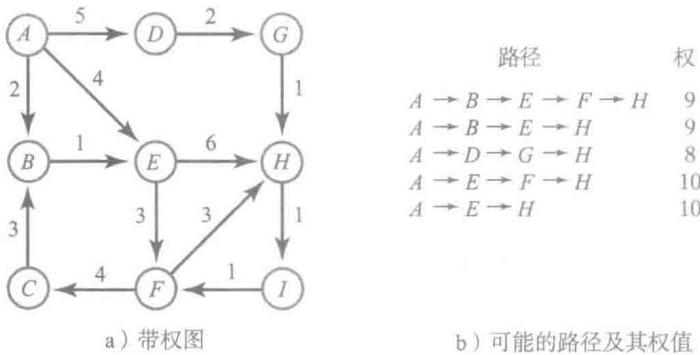


图 29-19 带权图及其从顶点 A 到顶点 H 间可能的路径



注：带权图中，两个给定顶点间的最短路径具有最小的边权值之和。找到这条路径的算法基于广度优先遍历。带权图中的几条路径可能有相同的最小边权值之和。我们的算法将找到这些路径中的一条。

优先队列中的每个项是一个对象，对象中含有一个顶点、从起始顶点到该顶点的路径代价及路径上的前驱顶点。优先级值是路径代价，最小值具有最高优先级。故最便宜路径位于优先队列的队头，所以最先被删除。注意，优先队列的多个项中可能含有相同的顶点但有不同的代价。

从算法得到的结果可知，图中的顶点包含了其前驱顶点及代价，这些信息能用来构造最便宜路径，与构造图 29-17 的最少边路径非常类似。

29.23

跟踪算法。图 29-20 跟踪了算法应用于图 29-19a 所示的带权图中，以顶点 A 为起始点的遍历部分。初始时，含值 A、0 及 null 的对象放入优先队列中。开始循环，从优先队列队头删除一项。使用这个项中的内容来修改图中所指顶点——目前是 A——的状态。故在 A 中路径长度存为 0，前驱顶点存为 null。还需标记 A 为已访问。

顶点 A 有 3 个未被访问的邻居 B、D 和 E。从 A 到每个邻居的路径代价分别是 2、5 和 4。用这些代价及 A 作为前驱顶点来创建对象，并放到优先队列中。优先队列将这些对象排序，所以最便宜路径在最前面。

从优先队列中删除最前面的项。项中包含顶点 B，所以访问 B。还要在顶点 B 中保存路径代价 2 及它的前驱顶点 A。现在 B 有唯一的未被访问的邻居 E。路径 $A \rightarrow B \rightarrow E$ 的代价是路径 $A \rightarrow B$ 的代价再加上从 B 到 E 的边的权值。总代价是 3。将 E、代价 3 及前驱顶点 B 封装为一个对象并加入优先队列中。注意，优先队列中的两个对象都含有顶点 E，但最新的这个具有最便宜的路径。

再次从优先队列中删除队头项。项中包含顶点 E，所以访问它，将路径代价 3、E 的前驱顶点 B 保存到 E 中。顶点 E 有两个未被访问的邻居 F 和 H。到每个邻居的路径的代价是到 E 的路径代价再加上到邻居的边权值。两个新对象加入优先队列中。

从优先队列中删除的下一个对象中含有顶点 E，但因为 E 已经被访问了，所以忽略它。然后从优先队列中删除下一个对象。算法继续这个过程，直到访问目标顶点 H 时为止。

图 29-21 显示了对图 29-20 进行跟踪得到的图的状态。通过观察目标顶点 H，可以看到从 A 到 H 的最便宜路径的长度是 8。从 H 到 A 反向跟踪，可以看到路径是 $A \rightarrow D \rightarrow G \rightarrow H$ ，与我们在段 29.21 中标识出的一样。

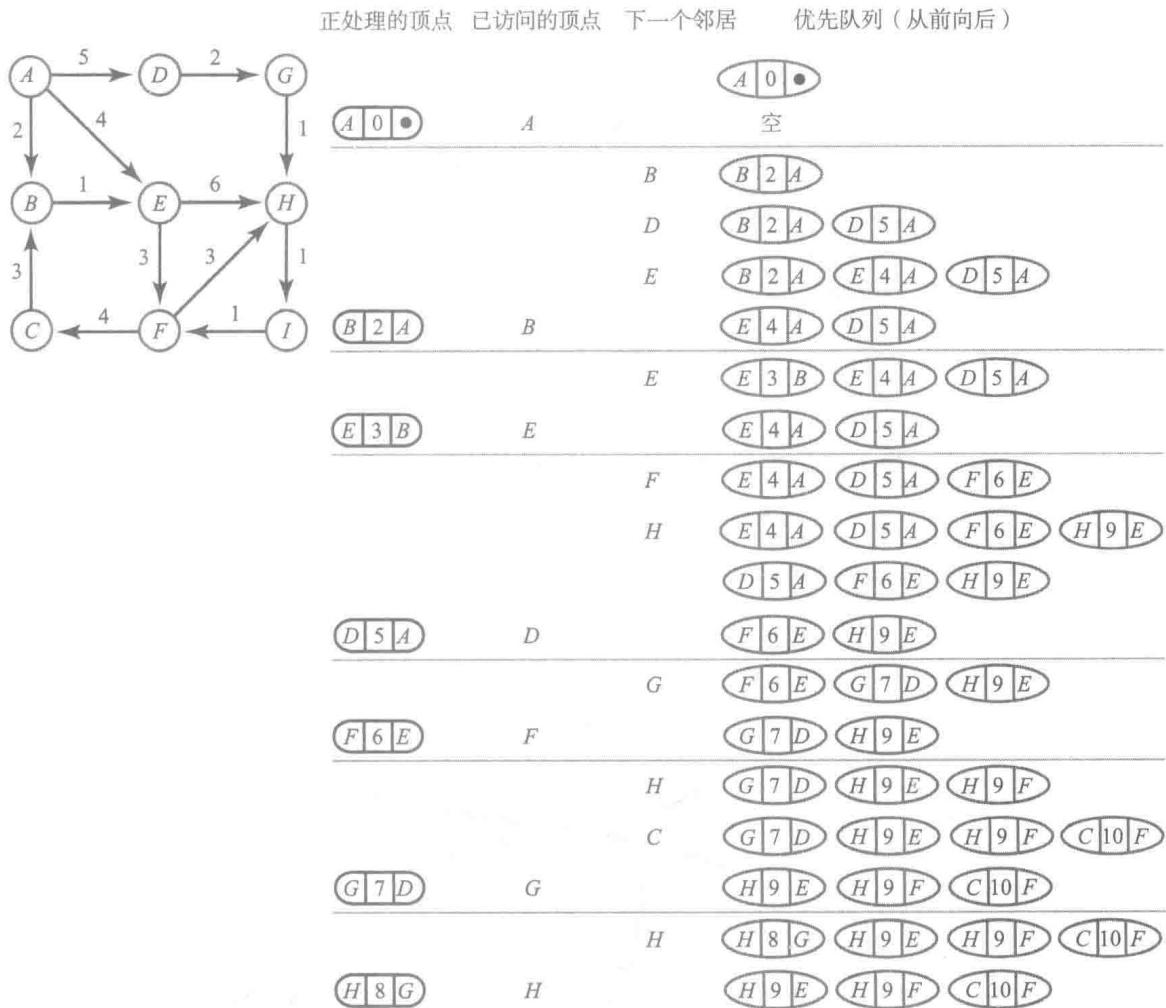


图 29-20 跟踪算法为得到带权图中从顶点 A 到顶点 H 间最便宜路径的遍历过程

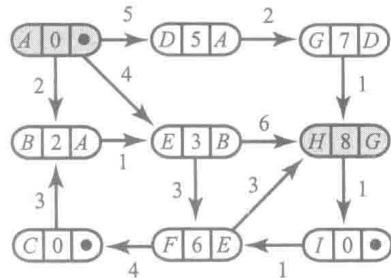


图 29-21 查找图 29-19a 中图从顶点 A 到顶点 H 间最便宜路径后的图

算法。刚刚描述的算法的伪代码如下所示。优先队列中的对象是私有类 EntryPQ 的实例。遍历过程中，算法将从 originVertex 到 endVertex 的最便宜路径上遇到的顶点，加入到所给的初始为空的栈 path 中。

```
Algorithm getCheapestPath(originVertex, endVertex, path)
done = false
priorityQueue=新的优先队列
priorityQueue.add(new EntryPQ(originVertex, 0, null))
```

```

while (!done && !priorityQueue.isEmpty())
{
    frontEntry = priorityQueue.remove()
    frontVertex=frontEntry 中的顶点

    if (frontVertex 未访问)
    {
        标记 frontVertex 为已访问
        将在 frontEntry 中记录的代价赋给到 frontVertex 的路径代价
        将在 frontEntry 中记录的前驱顶点赋给 frontVertex 的前驱顶点

        if (frontVertex 等于 endVertex)
            done = true
        else
        {
            while (frontVertex 有一个邻居)
            {
                nextNeighbor=frontVertex 的下一个邻居
                weightOfEdgeToNeighbor=到 nextNeighbor 的边权值

                if (nextNeighbor 未访问)
                {
                    nextCost=weightOfEdgeToNeighbor+到 frontVertex 的路径代价
                    priorityQueue.add(new EntryPQ(nextNeighbor, nextCost,
                        frontVertex))
                }
            }
        }
    }

    // Traversal ends; construct cheapest path
    pathCost = 到 endVertex 的路径代价
    path.push(endVertex)

    vertex = endVertex
    while (vertex 有前驱顶点)
    {
        vertex=vertex 的前驱顶点
        path.push(vertex)
    }
    return pathCost
}

```

最便宜路径的起始顶点位于栈 path 的栈顶。栈底是目标顶点。路径代价是算法的返回值。这个算法基于 Dijkstra 算法，Dijkstra 算法寻找从起始顶点到所有其他顶点的最短路径。



学习问题 10 修改图 29-20 中的跟踪过程，寻找从顶点 A 到顶点 C 的最短（最便宜）路径。

学习问题 11 为什么放入优先队列的是 EntryPQ 的实例而不是顶点？

用于 ADT 图的 Java 接口

29.25 ADT 图与其他的 ADT 有一点不同，一旦创建了图，便不用添加、删除或是获取成分。而是使用图来回答基于顶点间关系的问题。

我们将图的操作分为两个 Java 接口。第一个接口中的操作用来创建图并获得像顶点个数这样的基本信息。第二个接口中规范说明了本章前面所讨论的遍历及路径查找这样的操作。为方便起见，我们定义第三个接口 GraphInterface，它将前两个接口合在了一起。

为使这些接口尽可能一般化，在接口中规范说明图既可以是有向的也可以是无向的，既可以是带权的也可以是不带权的。第一个接口列在程序清单 29-1 中。泛型类型 T 表示标识

图中顶点的对象的数据类型。

程序清单 29-1 用于图基本操作的接口

```

1 package GraphPackage;
2 /**
3  * An interface of methods providing basic operations for directed
4  * and undirected graphs that are either weighted or unweighted.
5  */
6 public interface BasicGraphInterface<T>
7 {
8     /**
9      * Adds a given vertex to this graph.
10     * @param vertexLabel An object that labels the new vertex and is
11     * distinct from the labels of current vertices.
12     * @return True if the vertex is added, or false if not.
13     */
14     public boolean addVertex(T vertexLabel);
15
16     /**
17      * Adds a weighted edge between two given distinct vertices that
18      * are currently in this graph. The desired edge must not already
19      * be in the graph. In a directed graph, the edge points toward
20      * the second vertex given.
21      * @param begin An object that labels the origin vertex of the edge.
22      * @param end An object, distinct from begin, that labels the end
23      * vertex of the edge.
24      * @param edgeWeight The real value of the edge's weight.
25      * @return True if the edge is added, or false if not.
26      */
27     public boolean addEdge(T begin, T end, double edgeWeight);
28
29     /**
30      * Adds an unweighted edge between two given distinct vertices
31      * that are currently in this graph. The desired edge must not
32      * already be in the graph. In a directed graph, the edge points
33      * toward the second vertex given.
34      * @param begin An object that labels the origin vertex of the edge.
35      * @param end An object, distinct from begin, that labels the end
36      * vertex of the edge.
37      * @return True if the edge is added, or false if not.
38      */
39     public boolean addEdge(T begin, T end);
40
41     /**
42      * Sees whether an edge exists between two given vertices.
43      * @param begin An object that labels the origin vertex of the edge.
44      * @param end An object that labels the end vertex of the edge.
45      * @return True if an edge exists.
46      */
47     public boolean hasEdge(T begin, T end);
48
49     /**
50      * Sees whether this graph is empty.
51      * @return True if the graph is empty.
52      */
53     public boolean isEmpty();
54
55     /**
56      * Gets the number of vertices in this graph.
57      * @return The number of vertices in the graph.
58      */
59     public int getNumberOfVertices();
60
61     /**
62      * Gets the number of edges in this graph.
63      * @return The number of edges in the graph.
64      */
65     public int getNumberOfEdges();
66
67     /**
68      * Removes all vertices and edges from this graph resulting in an empty graph.
69      */
70     public void clear();
71 }
72 // end BasicGraphInterface

```



示例。假定类 UndirectedGraph 实现了程序清单 29-1 中所给的 BasicGraphInterface 接口，且类在包 GraphPackage 中。下列语句创建了图 29-22 所示的图，这是图 29-6 的一部分：

```
BasicGraphInterface<String> airMap = new UndirectedGraph<>();
airMap.addVertex("Boston");
airMap.addVertex("Provincetown");
airMap.addVertex("Nantucket");
airMap.addEdge("Boston", "Provincetown");
airMap.addEdge("Boston", "Nantucket");
```

此时，

```
airMap.getNumberOfVertices()
```

返回 3，而

```
airMap.getNumberOfEdges()
```

返回 2。



图 29-22 图 29-6 所示航线图的一部分

学习问题 12 修改前面哪些 Java 语句，可以让 airMap 成为带权图？

29.27

本章之前讨论的算法用到了前面接口中没有规范说明的图的操作。虽然可以将这些操作添加到接口中，以便客户可以实现不同的算法，例如拓扑排序等，但我们没有选择这样做。而是决定，实现图算法的方法作为图类的一部分。程序清单 29-2 中的接口规范说明了这些方法。再次说明，标识图中顶点对象的数据类型由泛型类型 T 来表示。

程序清单 29-2 用于已存在图的操作的接口

```
1 package GraphPackage;
2 import ADTPackage.*; // Classes that implement various ADTs
3 /** An interface of methods that process an existing graph. */
4 public interface GraphAlgorithmsInterface<T>
5 {
6     /** Performs a breadth-first traversal of this graph.
7      * @param origin An object that labels the origin vertex of the traversal.
8      * @return A queue of labels of the vertices in the traversal, with
9          the label of the origin vertex at the queue's front. */
10    public QueueInterface<T> getBreadthFirstTraversal(T origin);
11
12    /** Performs a depth-first traversal of this graph.
13     * @param origin An object that labels the origin vertex of the traversal.
14     * @return A queue of labels of the vertices in the traversal, with
15         the label of the origin vertex at the queue's front. */
16    public QueueInterface<T> getDepthFirstTraversal(T origin);
17
18    /** Performs a topological sort of the vertices in a graph without cycles.
19     * @return A stack of vertex labels in topological order, beginning
20         with the stack's top. */
21    public StackInterface<T> getTopologicalOrder();
22
23    /** Finds the shortest-length path between two given vertices in this graph.
24     * @param begin An object that labels the path's origin vertex.
25     * @param end An object that labels the path's destination vertex.
26     * @param path A stack of labels that is empty initially;
27         at the completion of the method, this stack contains
28             the labels of the vertices along the shortest path;
29             the label of the origin vertex is at the top, and
30                 the label of the destination vertex is at the bottom.
31     * @return The length of the shortest path. */
32    public int getShortestPath(T begin, T end, StackInterface<T> path);
33}
```

```

34     /** Finds the least-cost path between two given vertices in this graph.
35      @param begin An object that labels the path's origin vertex.
36      @param end   An object that labels the path's destination vertex.
37      @param path   A stack of labels that is empty initially;
38                  at the completion of the method, this stack contains
39                  the labels of the vertices along the cheapest path;
40                  the label of the origin vertex is at the top, and
41                  the label of the destination vertex is at the bottom.
42      @return The cost of the cheapest path. */
43  public double getCheapestPath(T begin, T end, StackInterface<T> path);
44 } // end GraphAlgorithmsInterface

```

程序清单 29-3 中的接口组合了 BasicGraphInterface 和 GraphAlgorithmsInterface。

程序清单 29-3 用于 ADT 图的接口

```

1 package GraphPackage;
2 public interface GraphInterface<T> extends BasicGraphInterface<T>,
3                                         GraphAlgorithmsInterface<T>
4 {
5 } // end GraphInterface

```

示例。假定你想找到 Truro 和 Falmouth 间的最短道路。“最短道路”是指具有最少英里数的道路，而不是具有最少边数的路。可以使用类似于段 29.26 中那样的语句，先来创建图 29-3 中的图。然后可以使用方法 getCheapestPath 来解答问题。下列语句表示了如何执行这些步骤，及如何显示最短道路中的城市名。

```

GraphInterface<String> roadMap = new UndirectedGraph<>();
roadMap.addVertex("Provincetown");
roadMap.addVertex("Truro");
...
roadMap.addVertex("Falmouth");
roadMap.addEdge("Provincetown", "Truro", 10);
...
roadMap.addEdge("Hyannis", "Falmouth", 20);
StackInterface<String> bestRoute = new LinkedStack<>();
double distance = roadMap.getCheapestPath("Truro", "Falmouth", bestRoute);
System.out.println("The shortest route from Truro to Falmouth is " +
    distance + " miles long and " +
    "passes through the following towns:");
while (!bestRoute.isEmpty())
    System.out.println(bestRoute.pop());

```

29.28

注：ADT 图的操作能让你创建图并回答关于顶点间关系的一些问题。

 **学习问题 13** 前一个例子找到了两个城镇之间的最短道路。为什么我们调用的是 getCheapestPath 方法而不是 getShortestPath 方法？

本章小结

- 图是不同顶点和不同边的集合。每条边连接两个顶点。子图是图的一部分，自身也是一个图。
- 树是有层次关系和一个根的特殊的图，根是树中所有其他结点——即顶点——的祖先。

- 有向图中的每条边都有从一个顶点向另一个顶点的方向。无向图中的边是双向的。
- 从一个顶点到另一个顶点的路径是边的序列。路径长度是这些边的数目。简单路径经过这其中的每个顶点一次。回路是开始及结束于同一顶点的路径。简单回路经过其他顶点一次。
- 带权图中的边具有称为权或代价的一个值。带权图中的路径有一个权或代价，是路径的边权值之和。
- 每对不同顶点间都有路径存在的图是连通图。每对不同顶点间都有边存在的图是完全图。
- 如果无向图中两个顶点由边相连，则它们称为邻接的。在有向图中，如果存在一条起始于 j 终止于 i 的有向边，则称顶点 i 与顶点 j 邻接。邻接顶点称为邻居。
- 使用深度优先遍历或广度优先遍历可以遍历图中的顶点。深度优先遍历沿图中一条越来越深的路径访问，然后再沿其他路径访问。广度优先遍历访问一个结点的所有邻居，然后访问邻居的邻居。
- 有向无环图隐含着顶点之间存在一种称为拓扑序的次序。这个次序不是唯一的。使用拓扑排序可找到这些次序。
- 可以使用图的深度优先遍历查看两个给定顶点间是否存在一条路径。
- 可以修改图的广度优先遍历，来寻找两个给定顶点间具最少边数的路径。
- 可以修改带权图的广度优先遍历，来寻找两个给定顶点间具最小代价的路径。

练习

1. 假定 5 个顶点位于虚拟五角大厦的角上。画出含这些顶点的连通图。
2. 使用段 29.1 ~ 段 29.4 中介绍的术语，描述图 29-23 中的每个图。

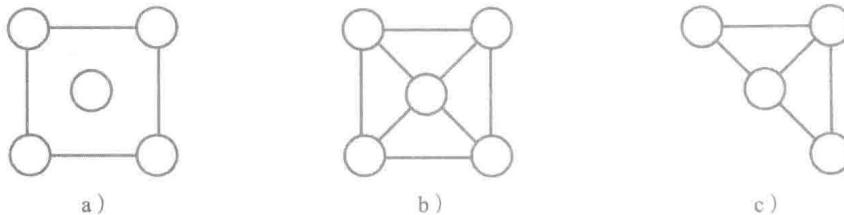


图 29-23 用于练习 2 的图

3. 考虑表示人之间相识关系的图。每个顶点表示一个人。每条边表示两个人之间的相识关系。
 - 这个图是有向图还是无向图？
 - 考虑与给定顶点 x 邻接的所有顶点。这个顶点集表示什么？
 - 这个图中的一条路径表示什么？
 - 什么环境下可能想知道这个图中两个顶点间的最短路径？
 - 与 2020 年 1 月 1 日活着的所有人对应的图，是连通图吗？证明你的答案。
4. 广度优先遍历访问图 29-10 中的顶点时，下列两种情况得到的次序是什么？
 - 初始顶点为 G
 - 初始顶点为 F
5. 使用深度优先遍历算法重做前一个练习。
6. 考虑图 29-10 中出现的有向图，删除顶点 E 和 F 之间的边，及顶点 F 和 H 之间的边。
 - 从顶点 A 开始广度优先遍历访问顶点的次序是什么？

- b. 使用深度优先遍历算法重做问题 a。
7. 画出表示你专业要学课程先修关系的有向图。找出这些课程的一个拓扑序。
8. 构造图 29-24 所示的带权有向无环图的拓扑序。
-
- 图 29-24 用于练习 8 和练习 22 的图
9. 像互联网或局域网这样的计算机网络可以表示为一个图。每台计算机是图中的一个顶点。两个顶点间的边表示两台计算机直连。解释，什么时候及为什么，会对下列任务感兴趣：
- 找到图中的一条路径
 - 找到从一个特定顶点到其他顶点间的多条路径
 - 找到从一个特定顶点到其他顶点间的最短路径
 - 查看图是否是连通图
10. 在修改深度优先遍历算法的基础上，写出找到有向图中从顶点 a 到顶点 b 间路径的算法。段 29.16 概述了这个问题的解决方法。
11. 树是无环连通图。
- 从图 29-1 中最少删除多少条边可使其变为树？
 - 给出一个这样的边集示例。
12. 图 29-7b 是表示迷宫的图。给图中的顶点打标记，最上面的顶点标记为 S （迷宫的入口），最下面的顶点标记为 T （迷宫的出口）。
- 这个图是树吗？
 - 从 S 到 T 间的最短路径是什么？
 - 图中最长简单路径是什么？
13. 修改图 29-15a 中的无权有向图，增加从 D 到 H 的一条有向边。得到的图中，从 A 到 H 的所有路径中有两条是最短路径。这两条路径中的哪条路径是段 29.19 中的 `getShortestPath` 算法找到的？
14. 在前一个练习中，删除从 E 到 H 的有向边，而不是添加从 D 到 H 的有向边，重做一遍。
15. 修改图 29-19a 中的带权有向图，增加从 D 到 H 的一条有向边。新加边的权值为 3。得到的图中，从 A 到 H 的所有路径中有两条是最便宜路径。这两条路径的哪条路径是段 29.24 中的 `getCheapestPath` 算法找到的？
16. 找一张美国主要航空公司的航线图。这样的地图通常印在飞行杂志的背面。也可以从互联网上搜索到。地图很像是图 29-6 所示的图那样。考虑下列每对城市：
- Providence (RI) 和 San Diego (CA)
 - Albany (NY) 和 Phoenix (AZ)
 - Boston (MA) 和 Baltimore (MD)

- Dallas (TX) 和 Detroit (MI)
- Charlotte (NC) 和 Chicago (IL)
- Portland (ME) 和 Portland (OR)
- 列表中哪对城市间有边（直达航班）？
 - 哪对城市间没有任何路径相连？
 - 对其余的每对城市，找出最少边的路径。
17. 找到一张越野滑雪区的雪道地图。用无向图表示这张雪道图，其中雪道的每个交叉口都是一个顶点，交叉口间的每段雪道是一条边。考虑一位越野滑雪者，他希望滑一条最长的路但不想两次经过任一条雪道。起点及终点都在滑雪小屋，且不会两次经过任一段雪道的最长路径是什么？（交叉口可能经过多次，有些雪道可能没滑过。）
18. 找到一张速降滑雪区的雪道地图。用图表示这张雪道图，其中雪道的每个交叉口都是一个顶点，交叉口间的每段雪道是一条边。
- 这张图是有向图还是无向图？
 - 图中有环吗？
 - 找到起点在山顶、终点在滑雪小屋的最长路径。
19. 编写类 `UndirectedGraph` 的客户语句，创建图 29-3 所示的图。假定 `UndirectedGraph` 实现了 `GraphInterface`。
20. 编写类 `DirectedGraph` 的客户语句，创建图 29-8 所示的图。假定 `DirectedGraph` 实现了 `GraphInterface`。然后编写查找并显示图的拓扑序列的语句。
21. 如果图中每一对顶点间的两条路径不共享边或顶点，称图为双连通的（biconnected）。
- 图 29-1 和图 29-4 中的图哪个是双连通的？
 - 双连通图有哪些应用？
22. 带权有向无环图中的关键路径（critical path）是有最大权值的路径。假定所有的边权值都是正的。将到达一个顶点的路径权值赋给该顶点。初始时，每个顶点的值是 0。
- 按拓扑序每次检查一个顶点，可以找到关键路径。对每个顶点，考虑离开那个顶点的所有边。对其中的每条边，将边的权值与边源点的值相加。将这个和值与边目标点的值相比较。将较大的值作为目标顶点的值。访问过所有顶点后，顶点中保存的最大值即是关键路径的权。
- 找出图 29-24 所示图的关键路径。

项目

- 在查找树中很容易查找任何值。对其他的树，结点的孩子并没按某种特定方式有序，可以使用图中描述的广度优先遍历，来找到从根到其他结点（顶点） v 的一条路径。为一般树实现这样一个方法。
 - 写 Java 代码，创建图 29-1 所给的图。找到从 Sandwich 到 Falmouth 间的最短路径。对图 29-3 中的带权图也同样处理。（见练习 19。）
 - 写 Java 代码，创建图 29-10 所给的图。从结点 A 开始对图进行广度优先遍历。
 - 在 Nim 游戏中，任意数量的石块分为任意堆。每名玩家可以从一堆中移走任意数量的石块。移走最后一块石块的玩家获胜。
- 考虑这个游戏的受限版本，其中有 3 堆石块，分别含有 3、5 和 8 块石头。可以将这个游戏表示为一个有向图。图中的每个顶点对应堆的一种可能状态（每堆中的石块数）。例如，初始结构是 $(3, 5, 8)$ 。图中的每条边表示游戏中的一个合法移动。
- 写出构造这个有向图的 Java 语句。
 - 讨论计算机程序如何用这个图来玩 Nim 游戏。
- 无权图的直径（diameter）是图中所有顶点对间最短距离中的最大值。

- a. 给出计算图的直径的算法。
 - b. 就图中所含顶点和边而言，你算法的大 O 性能是多少？
 - c. 实现你的算法。
 - d. 讨论改进算法性能的可能办法。

6. 练习 22 描述了如何找到带权有向无环图中关键路径的方法。写出寻找关键路径的方法。可以假定，
测试图是否无环的方法已存在。

7. n -puzzle (n 块拼图) 是一个单人游戏，它使用一个正好容纳 $n+1$ 个方块的正方形或矩形框架。游戏
开始时， n 个块分别编号为 1 到 n ，随机放在框架内。框架内有一个空位置，目标是滑动这些块——
每次可水平或垂直滑动一块——直到它们按数字顺序排列为止，如图 29-25a 所示。这个求解状态针对的是使用 4×4 框架的 15-puzzle。

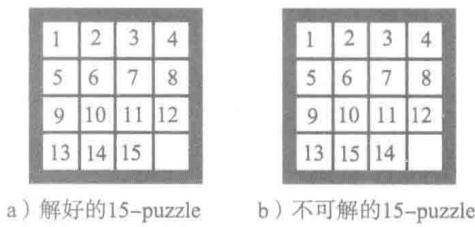


图 29-25 用于项目 7 的两个 15-puzzle

并不是所有的 n -puzzle 的初始状态都可解。例如，如果 15-puzzle 的初始状态如图 29-25b 所示，其仅需 14 和 15 两个块互换，就没有可行的解决方案。可解的 15-puzzle 最多用 80 步得到答案； 3×3 框架的 8-puzzle 如果可解，最多用 31 步解毕。为进一步减轻工作量，将考虑 2×3 框架的 5-puzzle。图 29-26 是这样的两个拼图及它们的答案。注意，答案中空位置可以在 1 的前面也可以在 5 的后面。

第 24 章图 24-23 是 tic-tac-toe 的游戏树。游戏树是含有特定游戏中可能的走步的一种图。因为不能改变 tic-tac-toe 中已走的步，所以游戏树是一个有向图。但对 n -puzzle 却不是这样，对任意走步都可以改变想法。所以用无向图来表示所有可能的走步。

编写 Java 代码，创建表示 5-puzzle 拼图状态的无向图。使用最短路径查找算法，找到任意给定初始状态的答案。

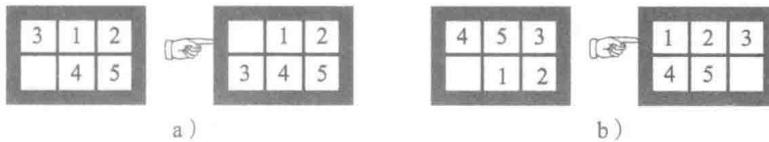


图 29-26 两个 5-puzzle 的初始和最终状态

第 30 章 |

Data Structures and Abstractions with Java, Fifth Edition

图的实现

先修章节：第 5 章、第 7 章、第 10 章、Java 插曲 4、第 13 章、第 20 章、第 24 章、第 29 章

目标

学习完本章后，应该能够

- 描述邻接矩阵
- 描述邻接表
- 规范说明并实现表示图中顶点和边的类
- 使用邻接表实现 ADT 图

与之前见过的 ADT 一样，图有几种实现方式。每种实现都必须表示图中的顶点及顶点间的边。一般地，使用线性表或是字典保存顶点，数组或是线性表用来表示边。边的每种表示各有千秋，但线性表表示是最经典的。

两个实现概述

ADT 图的两种常见实现使用数组或线性表来表示图中的边。数组通常是二维数组，称为邻接矩阵 (adjacency matrix)。线性表称为邻接表 (adjacency list)。每种结构都表示图中顶点间的连接——即边。

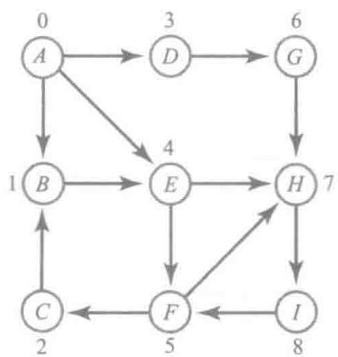
邻接矩阵

30.1 含 n 个顶点的图的邻接矩阵有 n 行 n 列。每行及每列对应图中的一个顶点。顶点编号为 0 到 $n-1$ ，从而与行下标和列下标相一致。如果 a_{ij} 是位于矩阵 i 行 j 列的元素，则 a_{ij} 表示顶点 i 和顶点 j 之间是否存在边。对于无权图，矩阵中可以使用布尔值。对于带权图，当边存在时可以使用边的权值，否则使用无穷大。

图 30-1 是无权有向图及其邻接矩阵的示例。现在来考虑图中的顶点 A ，我们将其编号为顶点 0。因为从顶点 A 到顶点 B 、 D 和 E 都存在有向边，所以矩阵元素 a_{01} 、 a_{03} 和 a_{04} 都为真。图中使用“T”来表示真。第一行中的其他项都是假（图中用空格表示）。

虽然从顶点 A 到顶点 B 间存在有向边，但反过来不成立。所以尽管 a_{01} 是真，但 a_{10} 是假。不过无向图的邻接矩阵是对称的 (symmetric)；即 a_{ij} 和 a_{ji} 有相同的值。当无向图从顶点 i 到顶点 j 之间存在边时，从顶点 j 到顶点 i 之间也存在边。

30.2 从邻接矩阵中，可以快速发现任给两个顶点间是否存在边。这个操作是 $O(1)$ 的。但如果你想知道某个特定顶点的所有邻居，必须扫描矩阵中的一整行，这是 $O(n)$ 操作。另外，矩阵占据了相当大的固定的空间，空间大小取决于顶点数但不依赖于边数。事实上，邻接矩阵表示图中每条可能的边，不管这条边是否存在。但是，大多数图中的边数相对较少——即它们是稀疏图。对于这样的图，邻接表表示使用的空间更少，下面就会看到这一点。



a) 图

| | A | B | C | D | E | F | G | H | I | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | T | | | T | T | | | | | 0 |
| B | | | | | T | | | | | 1 |
| C | T | | | | | | | | | 2 |
| D | | | | | | | T | | | 3 |
| E | | | | | | T | | T | | 4 |
| F | | T | | | | | | T | | 5 |
| G | | | | | | | T | | | 6 |
| H | | | | | | | | T | | 7 |
| I | | | | | T | | | | | 8 |

b) 图的邻接矩阵

图 30-1 无权有向图及其邻接矩阵



注：邻接矩阵使用固定大小的空间，空间大小取决于图中的顶点数但不依赖于边数。稀疏图的邻接矩阵浪费了空间，因为图中的边数相对较少。



注：使用邻接矩阵查看图中两个给定顶点间是否存在边是很快的。但如果你想知道某个顶点的所有邻居，必须扫描矩阵中的一整行。



学习问题 1 考虑第 29 章中的图 29-4b。从左上角开始，沿顺时针方向将顶点分别编号为 0 到 3。这个图的邻接矩阵是什么？

邻接表

给定顶点的邻接表仅表示以该顶点为起始点的边。在图 30-2 中，图 30-1a 中的每个顶点指向一个邻接点表。对于不存在的边不保留空间。所以邻接表合在一起，比起图 30-1b 中相应的邻接矩阵，使用更少的内存。因此，稀疏图的实现使用邻接表。本章给出的实现也将这样做，因为一般的图是稀疏的。30.3

虽然图中邻接表中包含的是顶点，但实际上在我们的实现中它们包含的是边。不过，这些边中的每一条，都将标注的顶点作为其终端顶点。



注：给定顶点的邻接表仅表示以该顶点为起始点的边。对于稀疏图，邻接表比邻接矩阵使用更少的内存。对于密集图，邻接矩阵可能是更好的选择。



注：使用邻接表时，通过遍历表可以找到某个顶点的所有邻居。如果想知道任意两个给定顶点间是否存在边，必须查找一个表。如果图中含有 n 个顶点，则最差情况下这些操作都是 $O(n)$ 的，但平均来讲要快一些。



学习问题 2 学习问题 1 中描述的图的邻接表是什么？

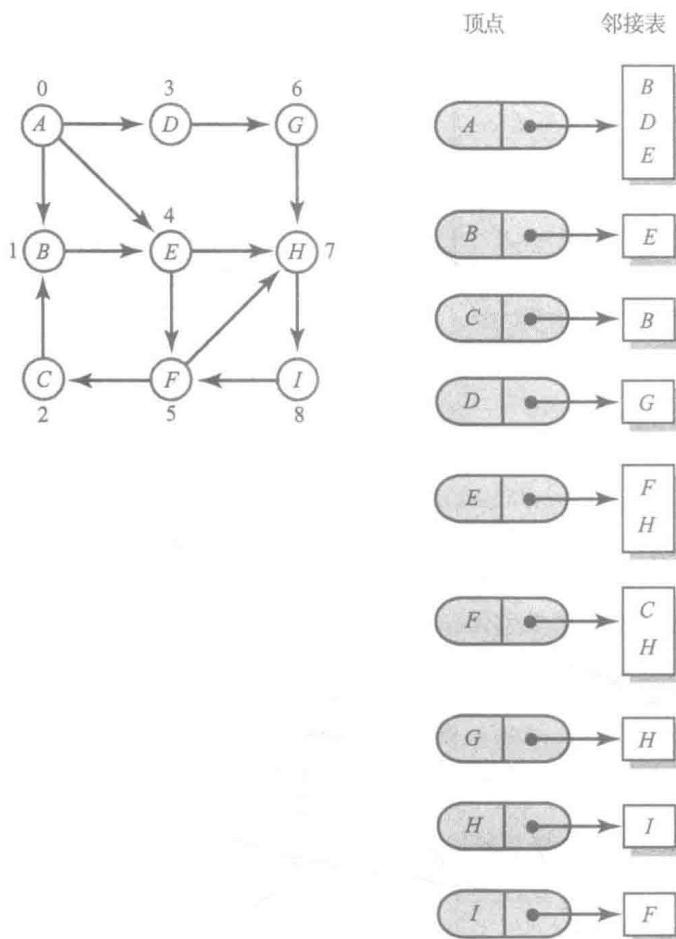


图 30-2 图 30-1a 中有向图的邻接表

顶点和边

30.4 设计实现 ADT 图的类时，遇到两种类型的对象，顶点和边。这些对象是相互关联的：顶点有从它出发的边，边由其两端的顶点定义。

图中的顶点有点类似于树中的结点。顶点和结点都是要对客户隐藏的实现细节。实现二叉树时，使用了包 – 友好的类 `BinaryNode`。（见第 25 章段 25.2。）这里，图对于类 `Vertex` 有包访问权限。前面我们在 `BinaryNode` 中不仅仅有简单的访问方法和赋值方法，从而简化了二叉树的实现。现在对 ADT 图的实现也是一样。实际上，在第 29 章（段 29.25）ADT 图的规范说明中，省略了实现不同图算法的必要操作。将这些操作赋给了顶点。

顶点结构比起二叉树中的结点结构，更像是一般树中结点的结构。图 25-8 中的一般结点和顶点都指向一个线性表，用它来说明其他的结点或顶点。

规范说明类 `Vertex`

30.5 标识顶点。首先，需要一种方法来标识图中的顶点。一个简单办法是使用整数或字符串。更一般的方法——在第 29 章使用过的——是用对象来标识每个顶点。这个标识是类 `Vertex` 的数据域。然后 `Vertex` 的操作可获取顶点的标识。我们使用构造方法设置标识，对这个域省略了赋值方法。

30.6

访问顶点。第 29 章讨论的算法需要在访问顶点时对其进行标注。所以，将标注顶点为已访问、测试一个顶点是否被访问及删除标注的操作都放在 `Vertex` 中。

邻接表。正如在本章前面提到过的，顶点的邻接表表示它的邻居。没有将邻接表放到图的类中，将它作为 `Vertex` 类的一部分会更方便。马上就会定义一个简单类 `Edge`，其实例将放在这些邻接表中。所以某个顶点的邻接表中含有从这个顶点发出去的边。每条边表示其权值（如果有）及边的终点。这样，`Vertex` 类需要方法，以便将边添加到邻接表中。这些方法本质上是将顶点与其邻居相连。

另外，必须能访问给定顶点的邻接表。所以定义一个返回顶点邻居的迭代器，以及返回与这些邻居间边的权值的迭代器。为方便起见，还包含了测试顶点是否至少有一个邻居的方法。

你会看到，邻接表是需要 `Edge` 实例的唯一地方。所以 `Edge` 类隐藏在 `Vertex` 内作为内部类，这是实现细节。

路径操作。寻找图的一条路径时，必须能定位路径上给定顶点之前的顶点——换句话说就是顶点的前驱顶点。所以，需要有对顶点前驱的设置、获取及测试操作。某些算法查找有最短长度的路径，或有最小权值或代价的路径。顶点可以记录下从起始点到自身的路径的长度或权值。所以，有操作来设置及获取记下的这个值。

Java 接口。程序清单 30-1 中的接口规范说明了我们刚介绍的顶点的操作。泛型 `T` 表示标识顶点对象的数据类型。

程序清单 30-1 用于图中顶点的接口

```

1 package GraphPackage;
2 import java.util.Iterator;
3 public interface VertexInterface<T>
4 {
5     /** Gets this vertex's label.
6      * @return The object that labels the vertex. */
7     public T getLabel();
8
9     /** Marks this vertex as visited. */
10    public void visit();
11
12    /** Removes this vertex's visited mark. */
13    public void unvisit();
14
15    /** Sees whether this vertex is marked as visited.
16     * @return True if the vertex is visited. */
17    public boolean isVisited();
18
19    /** Connects this vertex and a given vertex with a weighted edge.
20     * The two vertices cannot be the same, and must not already
21     * have this edge between them. In a directed graph, the edge
22     * points toward the given vertex.
23     * @param endVertex A vertex in the graph that ends the edge.
24     * @param edgeWeight A real-valued edge weight, if any.
25     * @return True if the edge is added, or false if not. */
26    public boolean connect(VertexInterface<T> endVertex, double edgeWeight);
27
28    /** Connects this vertex and a given vertex with an unweighted edge.
29     * The two vertices cannot be the same, and must not already
30     * have this edge between them. In a directed graph, the edge
31     * points toward the given vertex.
32     * @param endVertex A vertex in the graph that ends the edge.

```

```

33     @return True if the edge is added, or false if not. */
34     public boolean connect(VertexInterface<T> endVertex);
35
36     /** Creates an iterator of this vertex's neighbors by following
37      all edges that begin at this vertex.
38      @return An iterator of the neighboring vertices of this vertex. */
39     public Iterator<VertexInterface<T>> getNeighborIterator();
40
41     /** Creates an iterator of the weights of the edges to this
42      vertex's neighbors.
43      @return An iterator of edge weights for edges to neighbors of this
44      vertex. */
45     public Iterator<Double> getWeightIterator();
46
47     /** Sees whether this vertex has at least one neighbor.
48      @return True if the vertex has a neighbor. */
49     public boolean hasNeighbor();
50
51     /** Gets an unvisited neighbor, if any, of this vertex.
52      @return Either a vertex that is an unvisited neighbor or null
53      if no such neighbor exists. */
54     public VertexInterface<T> getUnvisitedNeighbor();
55
56     /** Records the previous vertex on a path to this vertex.
57      @param predecessor The vertex previous to this one along a path. */
58     public void setPredecessor(VertexInterface<T> predecessor);
59
60     /** Gets the recorded predecessor of this vertex.
61      @return Either this vertex's predecessor or null if no predecessor
62      was recorded. */
63     public VertexInterface<T> getPredecessor();
64
65     /** Sees whether a predecessor was recorded for this vertex.
66      @return True if a predecessor was recorded. */
67     public boolean hasPredecessor();
68
69     /** Records the cost of a path to this vertex.
70      @param newCost The cost of the path. */
71     public void setCost(double newCost);
72
73     /** Gets the recorded cost of the path to this vertex.
74      @return The cost of the path. */
75     public double getCost();
76 } // end VertexInterface

```

内部类 Edge

30.10 我们提到，将 Edge 的实例放在顶点的邻接表中，来表示源自那个顶点的边。所以，每条边必须记录边的终点及边的权值（如果有）。记录边的权值是需要边类的唯一原因。对于无权图，只需将顶点放到邻接表中。不过使用了边的对象，我们就能将顶点类用于带权图及无权图中了。

因为 Vertex 是使用 Edge 的唯一的类，所以将 Edge 写为 Vertex 的内部类。程序清单 30-2 是 Edge 类的实现。其中有个数据域用于边可能有的权值。对于无权图，不再创建无权边的类，而是将这个域置为 0。注意 Edge 的第二个构造方法将边的权值置为 0。

程序清单 30-2 保护类 Edge，作为 Vertex 的内部类

```

1  protected class Edge
2  {

```

```

3   private VertexInterface<T> vertex; // Vertex at end of edge
4   private double weight;
5
6   protected Edge(VertexInterface<T> endVertex, double edgeWeight)
7   {
8       vertex = endVertex;
9       weight = edgeWeight;
10    } // end constructor
11
12  protected Edge(VertexInterface<T> endVertex)
13  {
14      vertex = endVertex;
15      weight = 0;
16  } // end constructor
17
18  protected VertexInterface<T> getEndVertex()
19  {
20      return vertex;
21  } // end getEndVertex
22
23  protected double getWeight()
24  {
25      return weight;
26  } // end getWeight
27 } // end Edge

```



注：内部类 Edge 的实例含有边的终点及边可能有的权值。虽然对无权图这不是必需的，但 Edge 能让我们对带权图和无权图使用同一个顶点类。

类 Vertex 的实现

类的框架。为让 Vertex 类对图的客户程序隐藏，故将它放在第 29 章介绍的包 Graph-Package 内。程序清单 30-3 列出了类的框架，展示了数据域及构造方法。ADT 线性表有用于邻接表 edgeList 的迭代器。我们选用了第 13 章段 13.9 中讨论过的 LinkedListWith-Iterator 的链式实现。

30.11

程序清单 30-3 类 Vertex 的框架

```

1 package GraphPackage;
2 import java.util.Iterator;
3 import java.util.NoSuchElementException;
4 import ADTPackage.*; // Classes that implement various ADTs
5 class Vertex<T> implements VertexInterface<T>
6 {
7     private T label;
8     private ListWithIteratorInterface<Edge> edgeList; // Edges to neighbors
9     private boolean visited; // True if visited
10    private VertexInterface<T> previousVertex; // On path to this vertex
11    private double cost; // Of path to this vertex
12
13    public Vertex(T vertexLabel)
14    {
15        label = vertexLabel;
16        edgeList = new LinkedListWithIterator<>();
17        visited = false;
18        previousVertex = null;
19        cost = 0;
20    } // end constructor
21

```

```

22 < Implementations of the vertex operations go here. >
23 .
24 .
25 protected class Edge
26 {
27     < See Listing 30-2. >
28 } // end Edge
29 } // end Vertex

```



注: 类 `Vertex` 的数据域有助于实现第 29 章介绍的那些算法。例如，在广度优先遍历查找从一个顶点到另一个顶点的最便宜路径时，会用到域 `previousVertex` 和 `cost`。

30.12

两个 connect 方法。程序清单 30-1 中 `VertexInterface` 规范说明的每个 `connect` 方法，都将边放到顶点的邻接表中。先实现用于带权图的方法，然后用它来实现无权图的方法。方法中大部分工作是为了防止重复添加图中已有的边，或是添加顶点自身到自身的边。一旦这些细节都完成，`connect` 只需调用 ADT 线性表的 `add` 方法来添加边。

```

public boolean connect(VertexInterface<T> endVertex, double edgeWeight)
{
    boolean result = false;
    if (!this.equals(endVertex))
    { // Vertices are distinct
        Iterator<VertexInterface<T>> neighbors = getNeighborIterator();
        boolean duplicateEdge = false;
        while (!duplicateEdge && neighbors.hasNext())
        {
            VertexInterface<T> nextNeighbor = neighbors.next();
            if (endVertex.equals(nextNeighbor))
                duplicateEdge = true;
        } // end while
        if (!duplicateEdge)
        {
            edgeList.add(new Edge(endVertex, edgeWeight));
            result = true;
        } // end if
    } // end if
    return result;
} // end connect
public boolean connect(VertexInterface<T> endVertex)
{
    return connect(endVertex, 0);
} // end connect

```

虽然添加到线性表的操作可以是 $O(1)$ 时间的，但扫描表防止添加重复边却需要花费更多的时间。因为含 n 个顶点的图中的每个顶点都可能是最多 $n-1$ 条边的起始点，故 `connect` 操作是 $O(n)$ 的。但对于稀疏图，起始于任何顶点的边数远小于 n 。这种情形下，`connect` 明显快于 $O(n)$ 。

30.13

迭代器。方法 `getNeighborIterator` 返回顶点的邻接点即其邻居的迭代器。我们在类 `Vertex` 内定义了一个私有的内部类 `NeighborIterator`，用来实现 Java 的接口 `Iterator`。得到的 `getNeighborIterator` 的实现如下：

```

public Iterator<VertexInterface<T>> getNeighborIterator()
{
    return new NeighborIterator();
} // end getNeighborIterator

```

类 NeighborIterator 列在程序清单 30-4 中。它的构造方法建立了 LinkedListWithIterator 中定义的迭代器实例。方法 next 用这个迭代器遍历顶点邻接表中的边。然后，使用 Edge 中的方法 getEndVertex, next 访问邻居顶点并返回它。

程序清单 30-4 作为 Vertex 的内部类的私有类 NeighborIterator

```

1  private class NeighborIterator implements Iterator<VertexInterface<T>>
2  {
3      private Iterator<Edge> edges;
4
5      private NeighborIterator()
6      {
7          edges = edgeList.getIterator();
8      } // end default constructor
9
10     public boolean hasNext()
11     {
12         return edges.hasNext();
13     } // end hasNext
14
15     public VertexInterface<T> next()
16     {
17         VertexInterface<T> nextNeighbor = null;
18         if (edges.hasNext())
19         {
20             Edge edgeToNextNeighbor = edges.next();
21             nextNeighbor = edgeToNextNeighbor.getEndVertex();
22         }
23         else
24             throw new NoSuchElementException();
25         return nextNeighbor;
26     } // end next
27
28     public void remove()
29     {
30         throw new UnsupportedOperationException();
31     } // end remove
32 } // end NeighborIterator

```

用类似的方式，方法 getWeightIterator 返回私有内部类 WeightIterator 的实例。这个类类似于 NeighborIterator 类。

hasNeighbor 和 getUnvisitedNeighbor 方法。方法 hasNeighbor 使用 LinkedListWithIterator 中的 isEmpty 方法，测试 edgeList 是否为空：

```

public boolean hasNeighbor()
{
    return !edgeList.isEmpty();
} // end hasNeighbor

```

使用 getNeighborIterator 返回的迭代器，方法 getUnvisitedNeighbor 能返回尚未访问的邻接顶点。这项任务必须按拓扑序执行。

```

public VertexInterface<T> getUnvisitedNeighbor()
{
    VertexInterface<T> result = null;
    Iterator<VertexInterface<T>> neighbors = getNeighborIterator();
    while (neighbors.hasNext() && (result == null) )
    {
        VertexInterface<T> nextNeighbor = neighbors.next();
        if (!nextNeighbor.isVisited())

```

```

        result = nextNeighbor;
    } // end while
    return result;
} // end getUnvisitedNeighbor

```

30.15

其余的方法。Vertex类将重写equals方法。如果两个顶点的标识相同，则它们是相等的。

```

public boolean equals(Object other)
{
    boolean result;
    if ((other == null) || (getClass() != other.getClass()))
        result = false;
    else
    { // The cast is safe within this else clause
        @SuppressWarnings("unchecked")
        Vertex<T> otherVertex = (Vertex<T>)other;
        result = label.equals(otherVertex.label);
    } // end if
    return result;
} // end equals

```

Vertex的其余方法都不难实现，将它们留作练习。



学习问题3 给定接口VertexInterface和类Vertex，写Java语句，创建下列带权有向图中的顶点和边。这个图含有3个顶点，分别是A、B和C，4条边如下：
 $A \rightarrow B$ 、 $B \rightarrow C$ 、 $C \rightarrow A$ 和 $A \rightarrow C$ 。这些边的权值分别是2、3、4和5。

ADT图的实现

现在考虑如何使用Vertex来实现带权或无权的有向图。

基本操作

30.16

类的开始。不论是使用邻接表——此处所用的——还是邻接矩阵来实现，都必须有一个用于图中顶点的容器。如果使用整数标识顶点，则线性表就是这个容器的不二选择，因为每个整数都能对应到表中的一个位置。如果使用字符串这样的对象来标识它们，则字典是更好的选择。我们用的是这个办法。



注：不管图是哪种图，也不论你如何实现它，都需要字典这样的容器来保存图中的顶点。

一个小的有向图中顶点的字典如图30-3所示。顶点A和D都有从该点起始的边的邻接表。这些边中的字符表示指向字典中对应顶点的引用。因为ADT字典含有关键字-值对，所以可以使用顶点标识作为查找关键字，顶点本身作为对应的值。这个结构能快速定位到所给标识的具体顶点。

类的开始部分列在程序清单30-5中。回忆一下，泛型T表示图中标识顶点的对象的数据域。类的第一个数据域是顶点的字典。顶点数不是必需的，因为字典可以为我们统计顶点数。

因为每个顶点维护自己的邻接表，故图中的边不容易统计。所以，将边数保存在图类的一个数据域中。

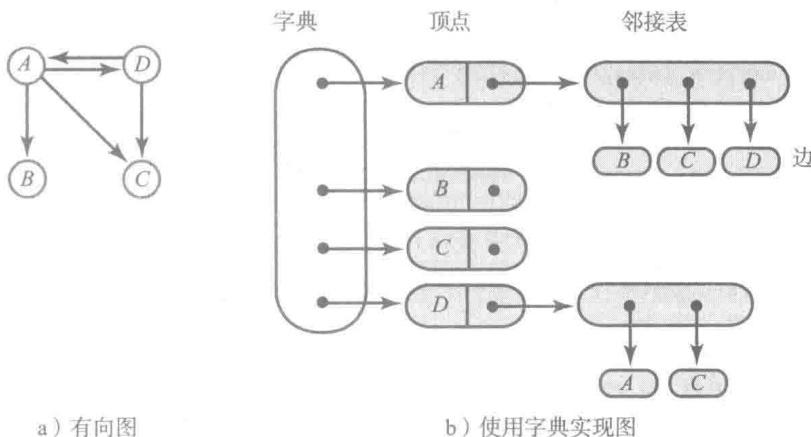


图 30-3 使用顶点字典的有向图的表示

程序清单 30-5 类 DirectedGraph 的框架

```

1 package GraphPackage;
2 import java.util.Iterator;
3 import ADTPackage.*; // Classes that implement various ADTs
4 public class DirectedGraph<T> implements GraphInterface<T>
5 {
6     private DictionaryInterface<T, VertexInterface<T>> vertices;
7     private int edgeCount;
8
9     public DirectedGraph()
10    {
11        vertices = new LinkedDictionary<T>();
12        edgeCount = 0;
13    } // end default constructor
14
15    < Implementations of the graph operations go here. >
16    .
17 } // end DirectedGraph

```

添加顶点。方法 addVertex 使用 Vertex 的构造方法创建新顶点。然后通过调用字典的方法 add，将顶点添加到字典中：

```

public boolean addVertex(T vertexLabel)
{
    VertexInterface<T> addOutcome =
        vertices.add(vertexLabel, new Vertex<T>(vertexLabel));
    return addOutcome == null; // Was addition to dictionary successful?
} // end addVertex

```

注意 vertexLabel 是用于字典项的查找关键字，新顶点是对应的值。回忆第 20 章段 20.4 中的接口，如果向字典的添加是成功的，则 add 返回 null。我们用这个事实来判定 addVertex 的返回值。

添加边。像 addEdge 这样通过标识识别已有顶点的方法，必须在字典 vertices 中查找顶点。为此，它们调用字典方法 getValue，使用顶点标识作为查找关键字。找到要被添加的边的两个顶点后，addEdge 将边添加到起始顶点的邻接表中。调用 Vertex 的 connect 方法来做这件事。如果边添加成功，则 edgeCount 加 1。图的 addEdge 方法定义如下，一个用于带权图，一个用于无权图：

30.17

30.18

```

public boolean addEdge(T begin, T end, double edgeWeight)
{
    boolean result = false;
    VertexInterface<T> beginVertex = vertices.getValue(begin);
    VertexInterface<T> endVertex = vertices.getValue(end);
    if ( (beginVertex != null) && (endVertex != null) )
        result = beginVertex.connect(endVertex, edgeWeight);
    if (result)
        edgeCount++;
    return result;
} // end addEdge
public boolean addEdge(T begin, T end)
{
    return addEdge(begin, end, 0);
} // end addEdge

```

30.19 边的测试。方法 hasEdge 的开头很像 addEdge，也要查找所需边的两个顶点。找到起始顶点，hasEdge 调用 Vertex 的方法 getNeighborIterator，在起始顶点的邻接表中查找所需的边。下列实现中，可以明白为什么将 equals 方法定义在 Vertex 中是这么重要。

```

public boolean hasEdge(T begin, T end)
{
    boolean found = false;
    VertexInterface<T> beginVertex = vertices.getValue(begin);
    VertexInterface<T> endVertex = vertices.getValue(end);
    if ( (beginVertex != null) && (endVertex != null) )
    {
        Iterator<VertexInterface<T>> neighbors =
            beginVertex.getNeighborIterator();
        while (!found && neighbors.hasNext())
        {
            VertexInterface<T> nextNeighbor = neighbors.next();
            if (endVertex.equals(nextNeighbor))
                found = true;
        } // end while
    } // end if
    return found;
} // end hasEdge

```

30.20 其他的方法。方法 isEmpty、clear、getNumberOfVertices 和 getNumberOfEdges 的实现很简单，如下所示。

```

public boolean isEmpty()
{
    return vertices.isEmpty();
} // end isEmpty
public void clear()
{
    vertices.clear();
    edgeCount = 0;
} // end clear
public int getNumberOfVertices()
{
    return vertices.getSize();
} // end getNumberOfVertices
public int getNumberOfEdges()
{
    return edgeCount;
} // end getNumberOfEdges

```

重置顶点。段 30.11 中看到，类 `Vertex` 有数据域 `visited`、`previousVertex` 和 `cost`。这些数据域是实现第 29 章中介绍的图算法所必需的。例如，一旦找到图的一条最短路径，图中很多顶点都被访问过了并已打上了标记。在同一图中执行拓扑排序之前，必须重置图中每个顶点的 `visited` 域。

下面的方法 `resetVertices` 将域 `visited`、`previousVertex` 和 `cost` 置为各自的初值。为此，方法用到接口 `DictionaryInterface` 中声明的一个迭代器。方法不是公有的，因为我们只在 `GraphAlgorithmsInterface` 所声明的方法中调用它。

```
protected void resetVertices()
{
    Iterator<VertexInterface<T>> vertexIterator = vertices.getValueIterator();
    while (vertexIterator.hasNext())
    {
        VertexInterface<T> nextVertex = vertexIterator.next();
        nextVertex.unvisit();
        nextVertex.setCost(0);
        nextVertex.setPredecessor(null);
    } // end while
} // end resetVertices
```

学习问题 4 对于学习问题 3 中描述的图，创建类 `DirectedGraph` 的一个实例。

效率。在图中添加顶点的操作是 $O(n)$ 的，因为顶点是添加到链式字典中的。添加一条边涉及从字典中取回两个顶点，然后调用 `Vertex` 的方法 `connect`。所以方法 `addEdge` 也是 $O(n)$ 的。类似地，`hasEdge` 是 $O(n)$ 的，因为它先从字典中取回两个顶点。然后遍历以第一个顶点为始点的边，来查看这些边是不是终止于第二个顶点。正如你看到的，这 3 个图操作的执行依赖于图中顶点的个数。`BasicGraphInterface` 中其余的方法都是 $O(1)$ 的。图 30-4 总结了这些结论。

| <code>addVertex</code> | $O(n)$ |
|----------------------------------|--------|
| <code>addEdge</code> | $O(n)$ |
| <code>hasEdge</code> | $O(n)$ |
| <code>isEmpty</code> | $O(1)$ |
| <code>getNumberOfVertices</code> | $O(1)$ |
| <code>getNumberOfEdges</code> | $O(1)$ |
| <code>clear</code> | $O(1)$ |

图 30-4 使用邻接表实现的 ADT 图基本操作的性能

图算法

广度优先遍历。第 29 章段 29.12 提出了从给定起始顶点开始，对非空图进行广度优先遍历的算法。回忆一下，遍历先访问起始顶点，然后是起始顶点的邻居。然后访问起始顶点邻居的每个邻居。遍历使用队列来保存已访问的顶点。遍历次序就是顶点入队列的次序。但因为算法必须从队列中删除顶点，故我们将遍历次序保存在第二个队列中。因为这第二个队列是返回给客户的，所以将顶点标识入队，而不是将顶点入队。记住，类 `Vertex` 对客户是不可用的。

`getBreadthFirstTraversal` 的下列实现严格遵从第 29 章给出的伪代码。参数 `origin` 是标识遍历起始顶点的对象。

```
public QueueInterface<T> getBreadthFirstTraversal(T origin)
{
    resetVertices();
    QueueInterface<T> traversalOrder = new LinkedQueue<>();
    QueueInterface<VertexInterface<T>> vertexQueue = new LinkedQueue<>();
```

```

VertexInterface<T> originVertex = vertices.getValue(origin);
originVertex.visit();
traversalOrder.enqueue(origin); // Enqueue vertex label
vertexQueue.enqueue(originVertex); // Enqueue vertex

while (!vertexQueue.isEmpty())
{
    VertexInterface<T> frontVertex = vertexQueue.dequeue();
    Iterator<VertexInterface<T>> neighbors = frontVertex.getNeighborIterator();
    while (neighbors.hasNext())
    {
        VertexInterface<T> nextNeighbor = neighbors.next();
        if (!nextNeighbor.isVisited())
        {
            nextNeighbor.visit();
            traversalOrder.enqueue(nextNeighbor.getLabel());
            vertexQueue.enqueue(nextNeighbor);
        } // end if
    } // end while
} // end while

return traversalOrder;
} // end getBreadthFirstTraversal

```

类似的执行深度优先遍历的方法的实现留作练习。



学习问题 5 写 Java 语句，对学习问题 4 所创建的图，显示从顶点 A 开始进行广度优先遍历得到的顶点序列。

30.24

最短路径。无权图中，从一个顶点到另一个顶点间所有路径中的最短路径是有最少边数的路径。找到这条路径的算法——如在第 29 章段 29.19 中所见——基于广度优先遍历。当访问顶点 v 时，将其标记为已访问，记下图中位于 v 之前的顶点 p ，还记下遍历到 v 的路径长度。将该路径长度和指向 p 的引用一起放入顶点 v 中。当遍历到达所需的目标顶点时，可以根据顶点中的数据构造最短路径。

方法 `getShortestPath` 的实现严格遵从第 29 章给出的伪代码。形参 `begin` 和 `end` 是标注为路径起始顶点和目标顶点的对象。第 3 个形参 `path` 是初始为空的栈。方法的结果是，栈中包含沿最短路径的顶点标识。方法返回该路径的长度。

```

public int getShortestPath(T begin, T end, StackInterface<T> path)
{
    resetVertices();
    boolean done = false;
    QueueInterface<VertexInterface<T>> vertexQueue = new LinkedQueue<>();
    VertexInterface<T> originVertex = vertices.getValue(begin);
    VertexInterface<T> endVertex = vertices.getValue(end);

    originVertex.visit();
    // Assertion: resetVertices() has executed setCost(0)
    // and setPredecessor(null) for originVertex

    vertexQueue.enqueue(originVertex);
    while (!done && !vertexQueue.isEmpty())
    {
        VertexInterface<T> frontVertex = vertexQueue.dequeue();
        Iterator<VertexInterface<T>> neighbors = frontVertex.getNeighborIterator();
        while (!done && neighbors.hasNext())
        {
            VertexInterface<T> nextNeighbor = neighbors.next();
            if (!nextNeighbor.isVisited())

```

```

{
    nextNeighbor.visit();
    nextNeighbor.setCost(1 + frontVertex.getCost());
    nextNeighbor.setPredecessor(frontVertex);
    vertexQueue.enqueue(nextNeighbor);
} // end if
if (nextNeighbor.equals(endVertex))
    done = true;
} // end while
} // end while

// Traversal ends; construct shortest path
int pathLength = (int)endVertex.getCost();
path.push(endVertex.getLabel());

VertexInterface<T> vertex = endVertex;
while (vertex.hasPredecessor())
{
    vertex = vertex.getPredecessor();
    path.push(vertex.getLabel());
} // end while

return pathLength;
} // end getShortestPath
}

```

对于带权图的 `getCheapestPath` 方法的实现留作练习。



学习问题 6 写 Java 语句，对学习问题 4 所创建的图，显示从顶点 A 到顶点 C 的最短路径上的顶点。还要显示该路径长度。

本章小结

- 给定顶点的邻接表含有指向顶点邻居的引用。
- 使用邻接表时，可以快速找到某个顶点的所有邻居。但是如果想知道任意两个给定顶点间是否存在边，则必须查找一个表。
- 邻接矩阵是一个二维矩阵，它表示图中的边。如果使用从 0 到 $n-1$ 为顶点进行编号，则矩阵 i 行 j 列的项表示顶点 i 和顶点 j 之间是否存在边。对于无权图，矩阵中可以使用布尔值。对于带权图，当边存在时可以使用边的权值，否则使用无穷大。
- 使用邻接矩阵，可以快速发现任给两个顶点间是否存在边。但如果你想知道某个顶点的所有邻居，则必须扫描矩阵的一整行。
- 每个邻接表仅表示以该顶点为起始点的边，但一个邻接矩阵为图中每条可能的边都保留空间。所以，当图是稀疏图时，邻接表比相应的邻接矩阵使用更少的内存。因此，一般的图实现使用邻接表。
- 实现邻接表的一种办法是让其成为类 `Vertex` 的数据域。这样，将类 `Edge` 的实例放在邻接表中，带权图和无权图就都可以表示了。`Edge` 的数据域包括边的终点及边的权值。`Vertex` 可在包内访问，而不是公有访问。`Edge` 是 `Vertex` 类的内部类。所以 `Vertex` 和 `Edge` 都对图的客户隐藏。
- 为有助于实现不同的图算法，可以标记类 `Vertex` 的实例是否被访问过。还可以记录到达该顶点的路径数据，例如路径上的前驱顶点和路径的代价。

练习

1. 第 29 章图 29-19a 的邻接矩阵是什么？

2. 第 29 章图 29-24a 的邻接矩阵是什么?
3. 第 29 章图 29-19a 的邻接表是什么?
4. 第 29 章图 29-14 的邻接表是什么?
5. 什么时候邻接矩阵与邻接表有同样的空间效率?
6. 假定你仅想测试两个给定顶点间是否存在边。采用邻接矩阵或邻接表时, 哪种方式的效率更高?
7. 假定你仅想找到某个顶点的所有邻接点。采用邻接矩阵或邻接表时, 哪种方式的效率更高?
8. 完成段 30.11 中类 `Vertex` 的实现。
9. 段 30.23 和段 30.24 中给出的方法 `getBreadthFirstTraversal` 和 `getShortestPath` 的大 O 表示是什么?
10. 实现方法 `getDepthFirstTraversal`。第 29 章的段 29.13 提供了这个方法的伪代码。其大 O 表示是什么?
11. 对带权图实现方法 `getCheapestPath`。这个方法的伪代码列在第 29 章的段 29.24 中。其大 O 表示是什么?
12. 画出表示类 `DirectedGraph` 和 `Vertex`, 及与其支持类之间关系的类图, 像 `LinkedDictionary` 那样的。
13. 顶点的出度 (out degree) 是以该顶点为起始顶点的边数。顶点的入度 (in degree) 是以该顶点为终点的边数。修改类 `DirectedGraph`, 让其能计算任意顶点的入度和出度。
14. 假定有带权有向图, 其中每个顶点的出度和入度最多为 4。(见前一个练习。) 如果图有 n 个顶点, 可以使用 n 行 4 列的数组来表示它。每一行表示图中的不同顶点。对应顶点 v 的行中的项是开始于 v 的边的终点。因为顶点的出度可以小于 4, 故一行中的某些项可能是 `null`。
下列每个操作的大 O 表示是多少?
 - 测试两个给定顶点是否邻接
 - 找出与给定顶点邻接的所有顶点
15. 如果图中顶点可以分为两组, 每条边从一组内的一个顶点连接到另一组内的一个顶点, 则图称为二部图 (bipartite)。第 29 章的图 29-1 包含了一个二部图。可以将 Sandwich、Hyannis、Orleans 和 Provincetown 放在 A 组, 而 Barnstable、Falmouth、Chatham 和 Truro 放在 B 组。每条边从 A 组内的一个顶点连到 B 组内的一个顶点。
 - 图 29-4、图 29-6 和图 29-7b 是二部图吗?
 - 如何用不同于正则图的方式实现二部图, 以便能利用二部图的特性?
16. 考虑有 n 个结点 e 条边的有向图, 其中 $0 \leq e \leq n^2$ 。
 - 用邻接矩阵表示图时, 使用大 O 表示, 下列每个操作的时间复杂度是多少?
 - 测试两个顶点是否由一条边相连
 - 找到给定顶点的后继顶点
 - 找到给定顶点的前驱顶点
 - 假定使用邻接表而不是邻接矩阵来表示图, 重复 a 中的问题。
17. 图中的圈 (loop) 是开始及结束于同一个顶点的一条边。图 30-5 显示了带权有向图中的一个圈。
 - 举一个例子, 说明允许有圈还是有用的。
 - 邻接矩阵和邻接表能表示支持圈的图吗?
18. 当图中两个顶点间由同方向的两条或多条边相连时, 出现多重边 (multiple edge)。图 30-6 显示的带权有向图中, D 到 B 之间有多重边。
 - 举例说明, 什么情况下多重边是有用的。
 - 邻接矩阵能表示无权的有多重边的图吗?
 - 邻接矩阵能表示带权的有多重边的图吗?

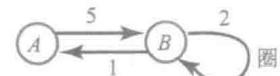


图 30-5 用于练习 17 的图

d. 邻接表能表示无权的多重边的图吗?

e. 邻接表能表示带权的多重边的图吗?

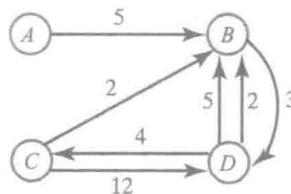


图 30-6 用于练习 18 的图

项目

1. 完成本章段 30.16 中类 `DirectedGraph` 的实现。
2. 从类 `DirectedGraph` 派生，实现无向图的类。哪些方法应该重写？`DirectedGraph` 中的哪些方法（如果有）不能用于无向图？如果这样的方法存在，新类中将如何处理？注意，方法 `getNumberOfEdges` 是对 `DirectedGraph` 中数据域的唯一访问方法。
3. 修改类 `DirectedGraph`，对数据域 `vertices` 和 `edgeCount` 定义保护的赋值方法。另外，对 `vertices` 定义保护的访问方法。然后使用修改的类 `DirectedGraph` 重做项目 2。比较本实现与项目 2 假设下可能的实现，对无向图执行 `addEdge` 方法的性能。
4. 使用邻接矩阵实现类 `Vertex` 和 `DirectedGraph`。
5. 假定有实现了无向图的类，实现检测无向图是否为无环图的方法。在广度优先遍历或深度优先遍历中，当发现一条边连到的顶点是已被访问的且不是前驱顶点时，查找到环。为简化最初的问题，可以假定图是连通的。然后去掉这个假设。
6. 实现检测图是否为连通图的方法。
7. 使用练习 14 中描述的表示，创建类 `LimitedVertex` 和 `LimitedDirectedGraph`。
8. 图的着色 (graph coloring) 为图中的每个顶点赋一个颜色，但同一种颜色不能赋给邻接点。如果能用 k 种或更少的颜色为图着色，则图称为 k - 色图 (k -colorable)。
给出算法，如果图是 2- 色图返回真，否则返回假。

练习 15 定义了二部图。说明图是二部图当且仅当它是 2- 色图。那么，基于这个事实，实现检测图是二部图的方法。

9. 重复第 11 章的项目 16，创建一个简单的社交网络。使用图来跟踪网络中人员之间的朋友关系。添加特性，让人能看到他朋友的朋友。
10. (游戏) 考虑第 1 章项目 9 描述的用于洞穴系统的 ADT。
 - a. 使用图来实现这个 ADT。
 - b. 使用 a 中定义的类，实现第 5 章项目 11 中要求你设计的算法。忽略那个项目中给出的提示。

附录 A |

Data Structures and Abstractions with Java, Fifth Edition

文档和程序设计风格

先修章节：Java 的一些知识

大多数程序会使用很多次，且会做某些修改，或是修改错误或是适应用户的新需求。如果程序不易读不易懂，那么它就不易修改。甚至不花很多精力都不能修改它。即使你的程序只使用一次，也应该在可读性上多费点心。毕竟，你调试程序时必须读懂它。

本附录中，我们讨论 3 个有助于提高程序可读性的技术：有意义的名字、缩进和注释。

命名变量和类

A.1 没有含义的名字几乎永远也不是好的变量名。给变量的名字应该表示变量的用途。如果变量用来对某事计数，或许可将它命名为 `count`。如果变量保存税率，或许可将它命名为 `taxRate`。

除了选择有意义及符合 Java 要求的名字外，还应该遵从其他程序员通常的做法。这样，当你参与多人合作的项目时，其他人更易读懂你的代码，且能将你的代码与他们的代码组合起来。按惯例，每个变量名都以一个小写字母开头，每个类名都以一个大写字母开头。如果名字含有多个字，则每个字的首字母大写，如变量名 `numberOfTries` 及类名 `StringBuffer`。

命名常量时使用全大写字母，以便与其他变量区分。如果有必要，各字之间使用下划线，如 `INCHES_PER FOOT`。

缩进

A.2 程序有一个结构：较小的部分含在较大的部分内。使用缩进来表示这个结构，这样可使程序易读。虽然 Java 忽略你使用的任何缩进，但统一的缩进是良好的程序设计风格所必不可少的一环。

每个类都从左边界开始，使用花括号括住它的定义。例如，可以写：

```
public class CircleCalculation
{
    .
}
```

`// end CircleCalculation`

数据域和方法在这些花括号内要缩进，如下列简单程序所示：

```
public class CircleCalculation
{
    public static final double PI = Math.PI;
    public static void main(String[] args)
    {
        double radius; // In inches
        double area;   // In square inches
    }
}
```

`// end main`

`} // end CircleCalculation`

在每个方法内，要缩进组成方法体的语句。这些语句也可能含有复合语句，它们缩进得更多。所以程序中有嵌套在语句中的语句。

嵌套的每一层应该比前一层缩进得更多一些，使得嵌套更清楚。最外层的结构完全不需要缩进。下一层要缩进。里面再嵌套的结构需要两倍缩进，以此类推。一般地，每层的缩进应该有 2 个或 3 个空格。要能清楚地看到缩进，但也要使用一行中的大部分地方来写 Java 语句。

如果语句不适合写在一行内，则可以将它写在 2 行或多行中。但是，当在多行中写一条语句时，后续的行应该比第一行缩进得更多，如下例所示：

```
System.out.println("The volume of a sphere whose radius is " +
    radius + " inches is " + volume +
    " cubic inches.");
```

归根到底，你必须遵从老师或项目经理给出的缩进规则——及通常的程序设计风格。任何情况下，在任何一个程序中都应该缩进一致。

注释

程序文档描述程序做什么及如何做。最好的程序是**自描述** (self-documenting) 的。即它们整洁的风格和精心挑选的名字，能让读程序的程序员明白程序的目的及逻辑。虽然你应该努力写这样的自描述程序，但你的程序可能还需要一些解释，以使它们能更加清晰。这些解释可以用**注释** (comment) 的形式给出。A.3

注释是程序中用来帮助人们理解程序的一些符号，但编译程序会忽略它们。许多文本编辑器自动用某种方式将注释标注出来，如用颜色显示它们。在 Java 中，注释有几种方式。

单行注释

要在一行写注释，注释的开头是双斜线 //。斜线后直到行尾间的所有内容都看作注释，并被编译程序忽略。这种形式对短注释是方便的，例如A.4

```
String sentence; // Spanish version
```

如果想将这种类型的注释写在多行中，则每行都必须含有符号 //。

注释块

写在一对符号 /* 和 */ 间的内容是注释，且被编译程序忽略。但这种形式一般不用来说明程序。相反，调试程序时暂时禁用一段 Java 语句用它是方便的。Java 程序员确实使用符号对 /** 和 */ 来标出某种形式的注释，如段 A.7 中所见。A.5

何时写注释

很难解释应该何时写注释。注释太多可能与注释太少一样不好。太多的注释可能将真正有用的内容给淹没了。太少的注释可能让读者对你自己很清楚的事情感到迷惑。记住，你也会读你自己的程序。如果下个星期读它，你能记清现在做的是什么吗？A.6

每个程序文件都应该由解释性注释开头。这个注释应该给出这个文件的所有重要信息：程序做什么，作者的名字，如何联系作者，文件最后的修改日期，课程中任务是什么。每个方法都应该以一个解释该方法的注释开头。

在方法内，必须用注释解释任何不明显的细节。注意，下面这样声明变量 `radius` 和 `area` 的注释是不好的：

```
double radius; // The radius
double area;   // The area
```

因为我们选择了描述性的变量名字，所以这些注释都是不言自明的。但不是简单地去掉这些注释，我们能写哪些不这样浅显的内容呢？半径使用的单位是什么？英寸？英尺？米？厘米？我们添加一条注释给出这些信息，如下：

```
double radius; // In inches
double area;   // In square inches
```

Java 文档注释

A.7 Java 语言还配有一个实用程序，名为 `javadoc`，用于生成描述类的 HTML 文档的。这些文档告诉程序的使用者如何来使用，但它们忽略所有的实现细节。

程序 `javadoc` 抽取类的头部、所有公有方法的头部及用特定形式写的注释。不抽取方法体及私有项。

要让 `javadoc` 抽取一条注释，则注释必须满足两个条件：

- 注释必须在公有类定义或公有方法头的前面。
- 注释必须以 `/**` 开头，以 `*/` 结尾。

段 A.12 中含有这种风格注释的示例。

你可以将 HTML 命令插人在注释中，这样能对 `javadoc` 控制得更多，但这不是必要的，本书中我们也没有这样做。

A.8 标签。为 `javadoc` 而写的注释通常含有特殊的标签（tag），标示出程序设计者及方法的形参及返回值等内容。标签以符号 `@` 开头。本附录中我们将仅介绍 4 种标签。

标签 `@author` 标出程序设计者的姓名，它出现在所有的类及接口中。一个标签后可以列出一个名字，或是用逗号分隔的几个名字。可以写几个这样的标签——每位作者用一个——或是完全忽略这个标签。因为由 `javadoc` 生成的文档忽略 `@author` 标签，所以有些组织并不使用它。

我们感兴趣的其他标签与方法一起使用。它们必须以下列次序出现在方法头部之前的注释中：

```
@param
@return
@throws
```

接下来我们介绍每个标签。

A.9 `@param` 标签。必须为方法内的每个形参写 `@param` 标签。应该按照形参在方法头部出现的次序列出这些标签。在 `@param` 标签后，要给出形参的名字及描述。一般地，使用短语而不是句子来描述这些参数，首先要提及参数的数据类型。参数名及描述之间不使用标点符号，创建它的文档时 `javadoc` 会插入一个破折号。

例如，注释

```
@param code      The character code of the ticket category.
@param customer The string that names the customer.
```

将在文档中生成下面的行：

code - The character code of the ticket category.
 customer - The string that names the customer.

@return 标签。必须为每个有返回值的方法写 **@return** 标签，即使你已经在方法的描述中描述了这个值。在这里要试着更具体地来描述这个值。这个标签必须接在注释中的 **@param** 标签之后。对于 **void** 方法及构造方法不要使用这个标签。

@throws 标签。接下来，如果方法可能抛出一个受检异常，则使用 **@throws** 标签来说明它，即使异常也出现在方法头的 **throws** 子句中。如果客户能合理地捕获，你也可以列出未检异常。(正如在附录 B 中所学的，类的客户是使用这个类的一个程序组件。) 为每个异常使用一个 **@throws** 标签，按名字的字典序列出它们。

 **示例。**下面是一个方法的 javadoc 注释示例。我们常常在这样的注释的开头对方法的目的做简要描述。这是我们的惯例；javadoc 没有用于这个目的的标签。

```
/** Adds a new entry to a roster.
 * @param newEntry      The object to be added to the roster.
 * @param newPosition   The position of newEntry within the roster.
 * @return True if the addition is successful.
 * @throws RosterException if newPosition < 1 or newPosition > 1 + the length
 *                         of the roster. */
public boolean add(Object newEntry, int newPosition) throws RosterException
```

javadoc 从前面这个注释中得到的文档如下所示：

add

```
public boolean add(java.lang.Object newEntry,
                   int newPosition)
                   throws RosterException
```

Adds a new entry to a roster.

Parameters:

newEntry - The object to be added to the roster.
 newPosition - The position of newEntry within the roster.

Returns:

True if the addition is successful.

Throws:

RosterException - if newPosition < 1 or newPosition > 1 + the length of
 the roster.

为节省本书的篇幅，有时我们会忽略实际程序中应该包含的注释部分。例如，有些方法可能只有目的描述，有些可能只有 **@return** 标签。注意，javadoc 接受这些缩写注释。

从 docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html 中可得到 javadoc 的更详细介绍。

附录 B |

Data Structures and Abstractions with Java, Fifth Edition

Java 类

先修章节：补充材料 1

本附录回顾了 Java 类、方法及包的使用和创建。即使你很熟悉这个材料，至少应该浏览一下以便了解我们的术语。

对象和类

B.1

补充材料 1（在线）介绍了类和对象的基础。回忆一下，一个对象包含数据及可以执行的确定动作。一个对象属于一个类，类中定义了它的数据类型。类规范说明了那个类所具有的对象的数据类型。类还规范说明了对象能执行的动作，及它们是如何完成那些动作的。面向对象程序设计（Object Oriented Programming, OOP）将程序视为由借助于动作而相互作用的对象组成的一种世界。例如，在模拟汽车的程序中，每辆汽车都是一个对象。

当我们在 Java 中定义一个类时，这个类就像是构建特定对象的一个计划或是蓝图。作为示例，图 B-1 描述了称为 `Automobile` 的类，并展示了 `Automobile` 的 3 个对象。这个类是对汽车是什么以及它能做什么的一般描述。

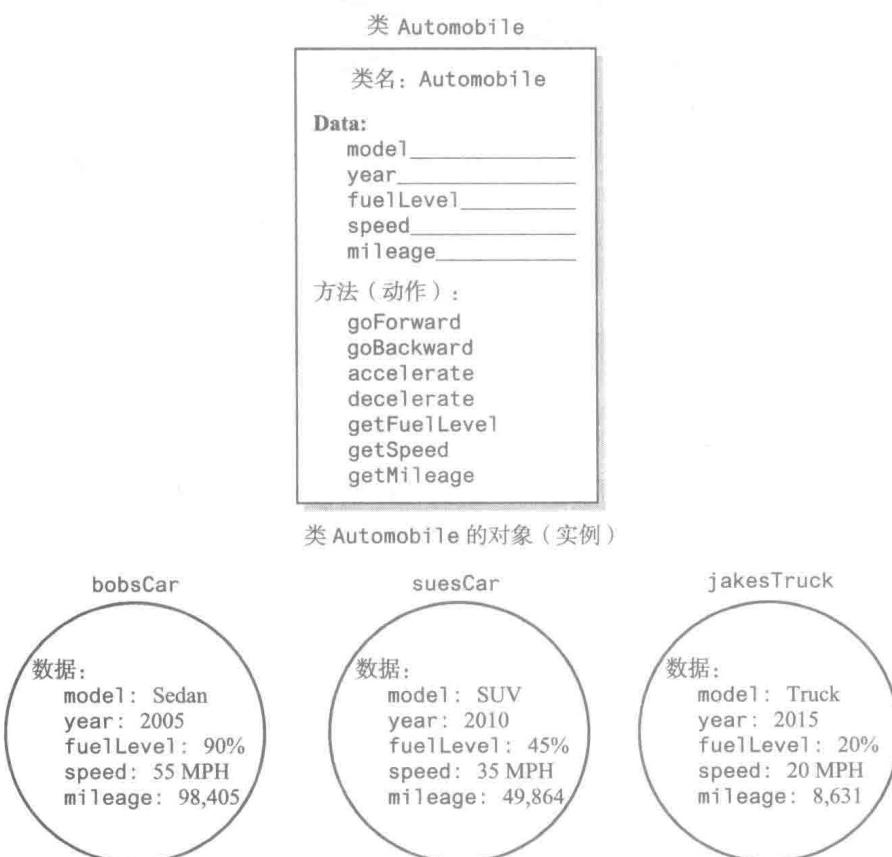


图 B-1 类的大纲及它的 3 个实例

类 `Automobile` 的每个实例 (instance), 或对象, 是一辆具体的汽车。你可以命名你所创建或实例化 (instantiate) 的每个对象。在图 B-1 中, 名字是 `bobsCar`、`suesCar` 和 `jakesTruck`。在 Java 程序中, `bobsCar`、`suesCar` 和 `jakesTruck` 应该是 `Automobile` 类型的变量。

`Automobile` 类的定义表明, `Automobile` 类的一个对象有车型、年份及油箱中有多少油这样的数据。类定义中不含有实际的数据——没有字符串也没有数值。各个对象具有数据, 而类只简单地指定它们拥有什么样的数据。

`Automobile` 类还定义了方法, 比如 `goForward` 和 `goBackward`。在使用 `Automobile` 类的程序中, `Automobile` 对象仅能执行那些方法中定义的动作。给定类的所有对象具有完全相同的数据。方法的实现指明动作是如何完成的, 及如何包含在类定义中的。不过, 实际上对象本身来执行方法的动作。

单个类中的对象可以具有不同的特性。即使这些对象有相同的数据类型和相同的数据, 各个对象的数据值也不同。

 注: 一个对象是含有数据并执行动作的一个程序结构。Java 程序中的对象相互作用, 这个相互作用组成了给定问题的求解方案。由对象执行的动作是由方法定义的。

 注: 一个类是对象的类型或种类。同一类中的所有对象有同类的数据及相同的数据。类定义是对象是什么及能做什么的一般描述。

 注: 编程时, 可以以几种不同的方式看待类。当你实例化类的一个对象时, 可以将类看作一种数据类型。当你实现一个类时, 可以将它看作构建对象的一个计划或蓝图——即作为对象数据及动作的定义。其他的时候, 可以将类看作有相同类型的对象集合。

在 Java 类中使用方法

我们假设有人想写一个称为 `Name` 的 Java 类, 用来表示一个人的名字。我们将描述如何使用这个类, 这个过程中, 将展示如何使用类的方法。使用类的程序称为类的客户 (client)。我们将保留“用户”一词用来表示使用程序的人。

例如, 要声明数据类型 `Name` 的一个变量, 可以这样写:

```
Name joe;
```

此时, 变量 `joe` 中什么都没有; 它还未初始化。要创建数据类型 `Name` 的一个具体对象——即要创建 `Name` 的一个实例——称为 `joe`, 可以写

```
joe = new Name();
```

`new` 运算符调用类中的一个特殊方法, 被称为构造方法 (constructor), 创建 `Name` 的一个实例。新对象的内存地址赋给 `joe`, 如图 B-2 所示。稍后在段 B.17 将展示如何定义构造方法。注意, 可以将前面两个 Java 语句合为一条:

```
Name joe = new Name();
```

假定人的名字仅有两部分: 名字和姓氏。对应于对象 `joe` 的数据则含有表示名字和姓氏的两个字符串。因为你



图 B-2 指向一个对象的变量

B.2

B.3

想能够设置 (set) ——即初始化或更改——一个人的名字，所以 `Name` 类中应该含有给你这个能力的方法。要设置 `joe` 的名字和姓氏，可以使用 `Name` 类的两个方法——`setFirst` 和 `setLast`，如下：

```
joe.setFirst("Joseph");
joe.setLast("Brown");
```

通常要调用一个方法时，先写接收对象 (receiving object) 的名字，后跟一个点，再写要调用的方法名，最后是一对包含实参 (argument) 的圆括号。本例中，`joe` 是接收对象，因为它接收对执行动作的调用，实参是代表输入给方法的字符串。方法将对象的数据域设置为实参中给定的具体值。

方法 `setFirst` 和 `setLast` 是 `void` 方法的示例，因为它们不返回值。正如在补充材料 1 (在线) 的段 S1.2 中提到过的，第二种方法——值方法——返回一个值。例如，方法 `getFirst` 返回一个字符串，是接收方法调用的对象的名字。类似地，方法 `getLast` 返回姓氏。

你可以在任何可以使用方法返回值类型的地方调用一个值方法。例如，`getFirst` 返回 `String` 类型的值，这样你可以在使用 `String` 类型值合法的任何地方，来使用如 `joe.getFirst()` 这样的方法调用。这些地方可能是一个赋值语句中，像是

```
String hisName = joe.getFirst();
```

或是 `println` 语句中，像是

```
System.out.println("Joe's first name is " + joe.getFirst());
```

注意到，方法 `getFirst` 和 `getLast` 在它们的括号中都没有实参。任何方法——值或 `void`——可以要求 0 个或多个实参。



注：值方法返回一个值；`void` 方法不返回值。例如，值方法 `getFirst` 返回代表名字的字符串。`void` 方法 `setFirst` 使用给定的字符串来设置名字，但不返回值。眼下，你可以通过对它们所做的事情的描述分辨值方法和 `void` 方法。过后，在段 B.7 到段 B.9 中，你会看到通过它们的 Java 定义来分辨它们。



学习问题 1 写 Java 语句，创建表示你名字的 `Name` 类型的对象。

学习问题 2 写 Java 语句，使用在学习问题 1 中创建的对象，以格式“名字 逗号 姓氏”显示你的名字。

学习问题 3 图 B-1 中给出的类 `Automobile` 中的哪些方法，最像是值方法，哪些方法最像是 `void` 方法？

引用和别名

B.4

Java 有 8 种基本数据类型：`byte`、`short`、`int`、`long`、`float`、`double`、`char` 和 `boolean`。基本类型的一个变量实际上含有一个基本值。所有其他的数据类型都是引用——即类或数组——类型。语句

```
String greeting = "Hello";
```

中的 `String` 类型变量 `greeting` 是一个引用变量。正如我们在补充材料 1 (在线) 中讨论过的，引用变量含有实际对象的内存地址。这个地址称为引用。知道 `greeting` 中含有的

是字符串 "Hello" 的引用而不是实际的字符串并不重要。这种情形下，更容易讨论字符串 greeting，实际上，这不是对那个变量的精确描述。当区分对象与对象的引用很重要时，本书中还是做了区分。

现在假定写了下面的语句

```
Name jamie = new Name();
jamie.setFirst("Jamie");
jamie.setLast("Jones");
Name friend = jamie;
```

两个变量 jamie 和 friend 指向同一个 Name 实例，如图 B-3 所示。我们说，jamie 和 friend 是别名，因为它们是同一个对象的两个不同的名字。当提及对象时，jamie 和 friend 可以互换着使用。

例如，如果使用变量 jamie 来修改 Jamie Jones 的姓氏，可以使用变量 friend 来访问它。所以语句

```
jamie.setLast("Smith");
System.out.println(friend.getLast());
```

显示 Smith。还注意到，布尔表达式 jamie==friend 为真，因为两个变量含有相同的地址。

定义一个 Java 类

现在展示如何写代表一个人名字的 Java 类 Name。将类定义保存在一个文件中，文件名是类名后跟 .java。所以类 Name 应该在文件 Name.java 中。一般地，每个文件中只保存一个类。

Name 对象中的数据含有作为字符串的一个人的名字和姓氏。类中的方法将能让你设置并查看这些字符串。类有下列格式：

```
public class Name
{
    private String first; // First name
    private String last; // Last name
    <此处是方法定义>
    .
}
// end Name
```

字 public 只意味着对使用该类的地方没有限制。即类 Name 在任何其他的 Java 类中都是可用的。两个字符串 first 和 last 称为类的数据域 (data field) 或实例变量 (instance variable) 或数据成员 (data member)。这个类的每个对象中都有这两个数据域。每个数据域声明前面的字 private 意味着只能在类内的方法中才可以通过它们的名字 first 和 last 使用它们。其他的类不能这样做。字 public 和 private 是访问修饰符 (access modifier) 或可见修饰符 (visibility modifier) 的示例，它们说明类、数据域或是方法可以使用的地方。还有第 3 个访问修饰符 protected，将在附录 C 中见到。



注：访问（可见）修饰符

字 public 和 private 是访问修饰符的示例，说明类、方法或数据域可以使用的地方。任何的类都可以使用公有方法，但私有方法只能用在定义它的类内。附录 C 讨论访问修饰符 protected，本附录的段 B.34 中将展示何时可以忽略访问修饰符。

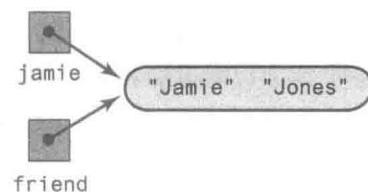


图 B-3 一个对象的别名

B.5

B.6 因为数据域是私有的，使用类 `Name` 的类如何能修改或查看它们的值？可以在类内定义方法，来查看或修改其数据域的值。可以将这样的方法声明为公有的，这样任何人都可以使用它们。能让你查看数据域值的方法称为访问方法（accessor method）或查询方法（query method）。修改数据域值的方法称为赋值方法（mutator method）。一般地，Java 程序员使用 `get` 作为访问方法名字的开头，使用 `set` 作为赋值方法名字的开头。因为这个惯例，访问方法有时也称为 `get` 方法或 `getter`，赋值方法称为 `set` 方法或 `setter`。例如，类 `Name` 有方法 `getFirst`、`getLast`、`setFirst` 和 `setLast`。

你可能认为访问方法和赋值方法使得让数据域私有的目的失效。事实正相反，它们让类有了对数据域的控制。例如，赋值方法可以检查对数据域的任何修改是不是合适的，如果有问题可以提出警告。如果类的数据域是公有的，类就不能做这个检查，因为任何人都能更改域。

 **注：**访问（查询）方法能让你查看数据域的值。赋值方法修改数据域的值。一般地，访问方法名的开头是 `get`，而赋值方法名的开头是 `set`。

 **程序设计技巧：**在类中每个数据域声明的开头应该写访问修饰符 `private`，让它们都是私有的。不能在类定义外直接引用私有数据域的名字。使用类的程序员强制只能通过类中的方法对数据域进行操作。而类能控制程序员如何访问或修改数据域。但在类的任何方法定义内，可以以你希望的方式使用数据域的名字。特别是，你可以直接修改数据域的值。

 **学习问题 4** 方法 `setFirst` 是访问方法还是赋值方法？

学习问题 5 一个典型的访问方法应该是值方法还是 `void` 方法？

学习问题 6 一个典型的赋值方法应该是值方法还是 `void` 方法？

学习问题 7 让类内的数据域是公有的，缺点是什么？

方法定义

B.7 方法定义有下列一般格式：

```
access-modifier use-modifier return-type method-name(parameter-list)
{
    method-body
}
```

使用修饰符（use modifier）是可选的，大多数情形下省略。当存在时，它可以是 `abstract`、`final` 或是 `static`。简单来说，抽象方法没有定义且必须要在派生类中被覆盖。终极方法不能在派生类中被覆盖。静态方法被类的所有实例共享。后面会遇到这些使用修饰符。

下一个返回类型（return type），它用于值方法，是方法返回值的数据类型。对于 `void` 方法，返回类型是 `void`。接下来写方法名及一对括号，内含可选的形参（parameter）列表和它们的数据类型。参数规范了要输入给方法的值或对象。

到此，我们描述了方法定义的第一行，这称为方法的头（header）或声明（declaration）。头之后是方法体（body）——它只是 Java 语句序列——包含在大括号中。

B.8 下面给出方法 `getFirst` 的定义，这是值方法的一个示例。

```
public String getFirst() ←头
{
    return first; } 体
} // end getFirst
```

这个方法返回数据域 `first` 中的字符串。所以这个方法的返回类型是 `String`。值方法必须永远执行一条 `return` 语句作为它的最后一个动作。返回值的数据类型必须与方法头中声明的数据类型相匹配。注意，这个具体方法没有形参。

现在来看一个 `void` 方法的示例。`void` 方法 `setFirst` 将数据域 `first` 设置为代表名字的字符串。方法的定义如下：

```
public void setFirst(String firstName)
{
    first = firstName;
} // end setFirst
```

这个方法不返回值，所以它的返回类型是 `void`。方法有一个形参 `firstName`，其数据类型是 `String`。它代表方法应该赋给数据域 `first` 的字符串。形参的声明永远含有一个数据类型和一个名字。如果有多个形参，则使用逗号分隔它们的声明。

对象 `this`。注意，前两个方法定义的方法体中通过名字用到了数据域 `first`。这完全合法。这里究竟涉及的是谁的数据域？记得这个类的每个对象都含有一个数据域 `first`。这里涉及的是属于接收方法调用的对象的数据域 `first`。当你想在方法定义的方法体内提到这个对象时，Java 有一个名字用于这个对象。它就是 `this`。例如在方法 `setFirst` 中可以将语句

```
first = firstName;
```

写为

```
this.first = firstName;
```

有些程序员以这种方式使用 `this`，要么是为了清晰，或是当他们想让形参与数据域同名时。例如，可以将 `setFirst` 的形参命名为 `first` 而不是 `firstName`。显然，方法体中的语句

```
first = first;
```

不能正确工作，所以你应该写

```
this.first = first;
```

通常，我们不会像这些例子中这样使用 `this`。不过段 B.26 将展示 `this` 的重要使用方式。



注：成员

数据域和对象的方法有时都称为对象的成员（member），因为它们属于对象。



注：命名类和方法

命名类和方法的惯例是，类名以大写字母开头，所有方法名以小写字母开头。使用名词或描述短语来命名一个类。使用动词或动作短语来命名一个方法。



注：局部变量

方法定义内声明的变量称为**局部变量**。局部变量的值在方法定义外不可用。如果两个方法有同名的局部变量，这两个变量是不同的，即使它们有相同的名字。

B.11

方法应该是一个独立单位。设计方法时应该与类中其他方法的附带细节分开，与使用这个类的程序分开。一个附带细节是方法形参的名字。幸好，形参的行为像是局部变量，所以它们的含义只局限于各自的方法定义内。故可以选择形参的名字，而不必担心它们与其他方法内使用的其他标识符相同。对于团队编程项目，一个程序员可能写一个方法定义，而其他程序员可能写使用这个方法的程序中的其他部分。两个程序员不需要在用于形参或局部变量的名字上达成一致。他们可以完全独立地选择自己的标识符，而不必担心他们的标识符中有部分或全部是一样的，或者完全没有相同的。

实参和形参

B.12

之前见过类的一个对象通常接收该类中定义的方法的调用。例如，你见过语句

```
Name joe = new Name();
joe.setFirst("Joseph");
joe.setLast("Brown");
```

设置 `joe` 对象的名字和姓氏。字符串 "Joseph" 和 "Brown" 是实参。这些实参必须对应于方法定义中的形参。例如，就 `setFirst` 而言，形参是字符串 `firstName`。实参是字符串 "Joseph"。实参与对应的形参对接上。所以在方法体中，`firstName` 代表字符串 "Joseph"，且表现得像是一个局部变量。

方法调用必须提供与对应的方法定义中的形参一样多的实参。另外，调用中的实参，就它们出现的次序及它们的数据类型，都必须对应于方法定义中的形参。但有时，当数据类型不匹配时，Java 将自动执行类型转换。

Java 中确实有一个符号用于可变数量的实参。因为我们确实不需要这个特性，所以不讨论它。



注：方法调用中的实参，就数量、次序及数据类型，都必须对应于方法定义中的形参。

旁白：术语“形参”和“实参”的使用

本书中术语“形参”和“实参”的使用与通常的用法是一致的，但有些人互换着使用这两个术语。有些人对我们所谓的“形参”和“实参”都使用“parameter”一词。另外一些人对我们所谓的“形参”和“实参”都使用“argument”一词。

传递实参

B.13

当形参是基本类型时，例如 `int` 或 `char`，形参被初始化为方法调用中对应实参的值。方法调用中的实参可以是字面值常量——像是 2 或 'A'——或者，它可以是能得到合适类型的值的一个变量或任何表达式。注意，方法不能改变有基本数据类型的实参的值。这样的实参只作为输入值使用。这种机制形容为按值调用（call-by-value）。

例如，假定类 `Name` 提供了由另一个数据域和方法 `setMiddleInitial` 定义的中间缩写字母。所以类的内容可能如下所示：

```
public class Name
{
    private String first;
    private char initial;
```

```

private String last;
...
public void setMiddleInitial(char middleInitial)
{
    initial = middleInitial;
} // end setMiddleInitial
...

```

这个类的客户可以含有如下的语句：

```

char joesMI = 'T';
Name joe = new Name();
...
joe.setMiddleInitial(joesMI);
...

```

图 B-4 展示了当方法 `setMiddleInitial` 执行时，实参 `joesMI`、形参 `middleInitial` 和数据域 `initial`。（虽然数据域有一个初值，但它不相关，所以图中将它显示为一个问号。）

如果方法改变了形参的值，对应的实参不会受到影响。例如，如果 `setMiddleInitial` 中含有语句

```
initial = middleInitial;
middleInitial = 'X'
```

则图 B-4c 中 `middleInitial` 的值将是 `X`，但图的其他部分不会改变。特别是 `joesMI` 的值不会改变。

当形参有类类型时，方法调用中对应的实参必须是那个类类型的对象。形参被初始化为那个对象的内存地址^①。所以，形参当作对象的别名使用。这隐含着如果对象有赋值方法，方法可以改变对象中的数据。但是方法不能用另一个对象替换实参对象。

例如，如果你收养一个孩子，可以让那个孩子跟着你的姓。假定你在类 `Name` 中添加了下面的方法 `giveLastNameTo`，修改这个名字：

```

public void giveLastNameTo(Name child)
{
    child.setLast(last);
} // end giveLastNameTo

```

注意，这个方法的形参是 `Name` 类型。

现在如果 Jamie Jones 收养了 Jane Doe，下面的语句将 Jane 的姓氏修改为 Jones：

```

public static void main(String[] args) {
    Name jamie = new Name();
    jamie.setFirst("Jamie");
    jamie.setLast("Jones");
    Name jane = new Name();
    jane.setFirst("Jane");
}

```



图 B-4 方法 `setMiddleInitial` 的执行对其实参 `joesMI`、形参 `middleInitial` 和数据域 `initial` 的影响效果

B.14

① 用于类类型形参的参数机制类似于按引用调用（call-by-reference）参数传递。如果你熟悉这项技术，要注意，在 Java 中类类型的形参与其他语言中的按引用调用有一点不同。

② 如果你不熟悉 `main` 方法和应用程序，请参阅补充材料 1（在线）的开头部分。

```
jane.setLast("Doe");
jamie.giveLastNameTo(jane);
}
// end main
```

图 B-5 展示在方法 `giveLastNameTo` 执行时的实参 `jane` 和形参 `child`。

B.15

如果你改变了方法定义，如下这样分配了一个新名字，将会如何？

```
public void giveLastNameTo2(Name child)
{
    String firstName = child.getFirst();
    child = new Name();
    child.setFirst(firstName);
    child.setLast(last);
} // end giveLastNameTo2
```

有了这个修改，则调用语句：

```
jamie.giveLastNameTo2(jane);
```

对 `jane` 没有影响，如图 B-6 所示。形参 `child` 就像是一个局部变量，所以它的值在方法定义之外不可用。

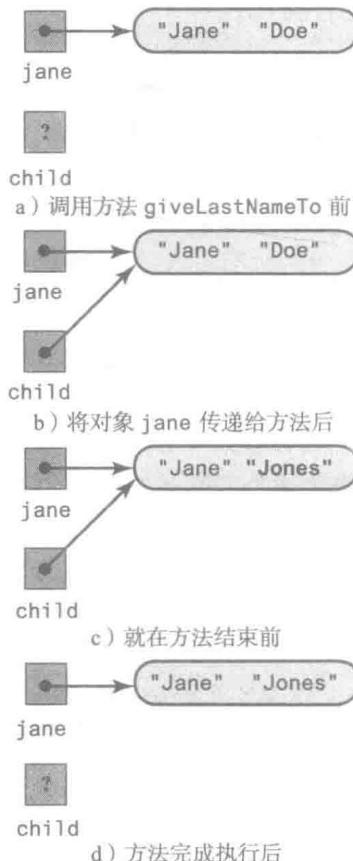


图 B-5 方法调用 `giveLastNameTo(jane)` 改变了作为实参传递给它的对象



学习问题 8 考虑由下列语句开头的方法定义

```
public void process(int number, Name aName)
```

如果 `jamie` 的定义如段 B.14 中那样，使用下面的语句调用这个方法

```
someObject.process(5, jamie);
```

传给方法定义内形参的值是什么？

学习问题 9 在学习问题 8 中，方法 process 能改变 jamie 中的数据域吗？

学习问题 10 在学习问题 8 中，方法 process 能给 jamie 赋值一个新对象吗？

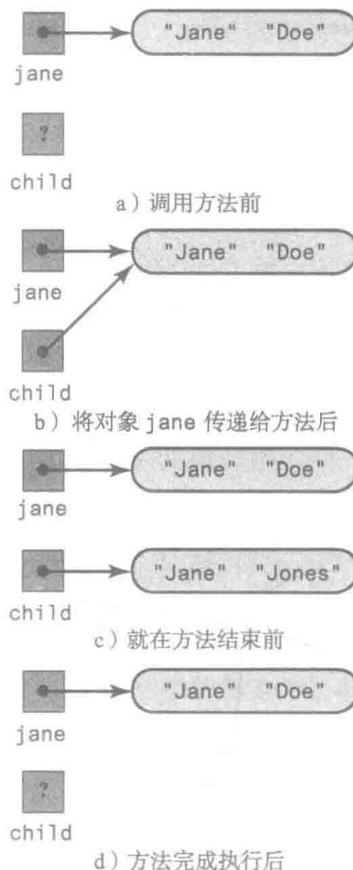


图 B-6 方法调用 `giveLastNameTo2(jane)` 不能替换作为实参传递给它的对象

Name 类的定义

程序清单 B-1 中给出了类 Name 的完整定义。我们通常将数据域声明放在类的开头，但有些人将它们放到最后。虽然 Java 允许方法定义与数据域声明混放一起，但我们更希望你不要那样做。B.16

下面各段将探讨这个类定义的一些细节。

程序清单 B-1 类 Name

```

1 public class Name
2 {
3     private String first; // First name
4     private String last; // Last name
5
6     public Name()
7     {
8     } // end default constructor
9

```

```

10  public Name(String firstName, String lastName)
11  {
12      first = firstName;
13      last = lastName;
14  } // end constructor
15
16  public void setName(String firstName, String lastName)
17  {
18      setFirst(firstName);
19      setLast(lastName);
20  } // end setName
21
22  public String getName()
23  {
24      return toString();
25  } // end getName
26
27  public void setFirst(String firstName)
28  {
29      first = firstName;
30  } // end setFirst
31
32  public String getFirst()
33  {
34      return first;
35  } // end getFirst
36
37  public void setLast(String lastName)
38  {
39      last = lastName;
40  } // end setLast
41
42  public String getLast()
43  {
44      return last;
45  } // end getLast
46
47  public void giveLastNameTo(Name aName)
48  {
49      aName.setLast(last);
50  } // end giveLastNameTo
51
52  public String toString()
53  {
54      return first + " " + last;
55  } // end toString
56 } // end Name

```

构造方法

B.17 段 B.2 中提到, 使用 `new` 运算符调用一个称为构造方法的特殊方法, 来创建一个对象。构造方法 (constructor) 为对象分配内存, 并初始化数据域。构造方法的方法定义有一些特性。一个构造方法

- 有与类一样的名字。
- 没有返回值, 甚至也没有 `void`。
- 有任意多个形参, 包括没有形参。

一个类可以有几个形参的类型及个数不同的构造方法。

不带形参的构造方法称为默认构造方法 (default constructor)。一个类只能有一个默认构造方法。`Name` 类的默认构造方法的定义如下

```
public Name()
{
} // end default constructor
```

具体到这个默认构造方法，它有一个空方法体，但它不必是空的。它可以显式地初始化数据域 `first` 和 `last` 的值，以不同于 Java 使用默认值所赋的值。

例如，可以如下定义构造方法：

```
public Name()
{
    first = "";
    last = "";
} // end default constructor
```

此处，我们将数据域 `first` 和 `last` 的值初始化为空字符串。如果构造方法有空方法体，则这些数据域将初始化为默认值 `null`。

 **注：**如果在构造方法中没有进行显式初始化，则数据域将被设置为默认值：引用类型的是 `null`，基本数值类型的是 0，布尔类型的是 `false`。

 **程序设计技巧：**如果类依赖于数据域的初值，则它的构造方法应该显式地设置这些值。
不要依赖于标准默认值。

 **程序设计技巧：**如果数据域有引用类型，则将它初始化为 `null` 之外的值。如果没做这步处理，当使用类时可能会引发 `NullPointerException` 异常。

如果你没有为类定义任何构造方法，那么 Java 会自动提供一个默认构造方法——即没有形参的构造方法。如果你定义了一个带形参的构造方法但没有定义默认构造方法——不带形参的——Java 将不会为你提供默认构造方法。因为类常常会反复重用，又因为最终你可能想创建一个不带指定形参的新对象，故你的类通常应包含一个默认构造方法。

 **注：**一旦你开始定义构造方法，Java 就不再为你定义任何构造方法。你定义的大多数的类应当含有一个默认构造方法。

类 `Name` 含有两个构造方法，当客户调用构造方法时，它将数据域初始化为给定的实参值：

```
public Name(String firstName, String lastName)
{
    first = firstName;
    last = lastName;
} // end constructor
```

这个构造方法有两个形参：`firstName` 和 `lastName`。要用这样的语句来调用它

```
Name jill = new Name("Jill", "Jones");
```

将名字和姓氏作为实参传给它。

在创建对象 `jill` 后，可以使用类的设置（赋值）方法改变其数据域的值。你看到，对于段 B.2 和段 B.3 中的对象 `joe` 来说，这个步骤是必要的，因为 `joe` 是由默认构造方法创建的，它用默认值——可能是 `null`——作为它的名字和姓。

让我们来看看，如果你试着使用构造方法来改变 `jill` 数据域的值时，会发生什么。当你创建对象后，变量 `jill` 含有那个对象的内存地址，如图 B-7a 所示。如果现在你写语句

B.18

B.19

B.20

```
jill = new Name("Jill", "Smith");
```

则创建了一个新对象，而 jill 含有它的内存地址。原来的对象丢失了，因为没有程序变量中保存它的地址，如图 B-7b 所示。

当程序中的变量不再指向一个内存位置时，它会怎样？Java 运行时环境会定期地释放 (deallocate) 这些内存位置，将它们返还给操作系统，这样它们可以再次被使用。实际上，内存被回收了。这个过程称为自动垃圾收集 (automatic garbage collection)。



a) 指向新创建对象的引用

b) 指向对象的引用丢失后对象被释放

图 B-7 一个对象和它的引用



注：内存泄漏

如果 Java 运行时环境不跟踪并回收程序不再引用的内存，则程序可能用到了它能够使用的所有内存，然后失败了。如果你使用另一种程序设计语言——例如 C++——则你要负责将不再需要的内存返还给操作系统以便下次再用。未能成功返还这样的内存的程序，会有所谓的内存泄漏 (memory leak)。Java 程序不会有这个问题。



注：没有赋值方法的类

创建没有设置方法的类的一个对象后，你不能改变其数据域的值。如果你需要改变，则必须使用构造方法创建一个新对象。Java 插曲 6 进一步讨论了这些类。



学习问题 11 什么是默认构造方法？

学习问题 12 如何调用一个构造方法？

学习问题 13 如果你没有为类定义构造方法会怎样？

学习问题 14 如果你没有定义默认构造方法但定义了一个带形参的构造方法，会怎样？

学习问题 15 当一个对象不再有一个变量指向它时会怎样？

toString 方法

B.21 类 Name 中的 `toString` 方法返回一个人全名的字符串。例如，你可以使用这个方法，写下面这个语句来显示对象 jill 表示的名字

```
System.out.println(jill.toString());
```

`toString` 方法不同寻常的地方是，当你写下面这个语句时，Java 会自动调用它

```
System.out.println(jill);
```

因此，为类提供方法 `toString` 通常是个好主意。如果你没有这样做，Java 会提供它自己的 `toString` 方法，这个方法产生一个对你没有什么意义的字符串。附录 C 中提供了关于 `toString` 方法的更多细节。

调用其他方法的方法

注意 Name 类定义中的方法 `setName`。虽然 `setName` 可以使用赋值语句来初始化 `first` 和 `last`，但它没有，而是调用方法 `setFirst` 和 `setLast`。因为这些方法是类的成员，所以 `setName` 可以调用它们，而不需要在名字前面加上对象变量及一个点。如果你愿意，可以使用 `this`，这个调用可以写为

```
this.setFirst(firstName);
```

当方法定义的逻辑复杂时，你应该将逻辑分为更小的块，并将每个块实现为一个独立的方法。然后你的方法可以调用这些其他的方法。但这些帮助方法或许不适合于客户使用。假如是这样，将它们声明为私有的而不是公有的，这样只有你自己的类才可以调用它们。

 **程序设计技巧：**如果一个帮助方法不适合公用，则将它声明为私有的。

类 Name 中的 `getName` 方法，还调用了 Name 中的另一个方法 `toString`。这里，我们想让 `getName` 和 `toString` 都返回相同的字符串。不是在两个方法中写相同的语句，而是让一个方法调用另一个。这能确保两个方法总是返回相同的值。以后，如果你修改了 `toString` 的定义，你将自动地修改 `getName` 返回的字符串。

 **程序设计技巧：**如果你想让两个方法有相同的行为，让其中一个调用另一个。

虽然让方法调用其他方法通常是避免重复代码的好主意，但如果从构造方法体中调用公有方法时，你必须要小心。例如，假定你想在段 B.19 中提到的构造方法中调用 `setName`。而从你的类派生的另一个类可以改变 `setName` 的效果，从而也改变了你构造方法的效果。一个解决办法是，定义一个让构造方法和 `setName` 都调用的私有方法。另一个解决办法，在附录 C 的段 C.18 中给出。

使用 this 来调用构造方法。你可以使用保留字 `this`，在构造方法体中调用另一个构造方法。例如，类 Name 有两个构造方法。段 B.16 中给出的默认构造方法有一个空方法体。段 B.17 提出，有一个默认构造方法来显式初始化类的数据域是个好主意，所以我们重写它，如下所示。

```
public Name()
{
    first = "";
    last = "";
} // end default constructor
```

我们修改默认构造方法，让它调用第二个构造方法来初始化 `first` 和 `last`，效果是一样的。如下所示。

```
public Name()
{
    this("", "");
} // end default constructor
```

语句

```
this("", "");
```

调用有两个形参的构造方法。按这种方式，初始化工作在一个地方进行。

! 程序设计技巧：使用 `this` 让一个构造方法调用另外一个构造方法，从而将几个构造方法的定义链接在一起。`this` 的使用必须位于构造方法定义的方法体的第一条。

? 学习问题 16 类 `Name` 中的第三个构造方法，可能有下面的方法头：

```
public Name(Name aName)
```

这个构造方法创建一个 `Name` 对象，其数据域与 `aName` 对象的各域值相等。通过调用已有的构造方法来实现这个新构造方法。

返回其类的实例的方法

B.26 类 `Name` 中的方法 `setName` 是一个 `void` 方法，它设置 `Name` 对象的名字和姓氏。我们可能这样使用这个方法：

```
Name jill = new Name();
jill.setName("Jill", "Greene");
```

这里，`setName` 设置了接收对象 `jill` 的名字和姓氏。

现在不是将 `setName` 定义为一个 `void` 方法，而是让它返回一个指向修改后的 `Name` 实例的引用，如下所示。

```
public Name setName(String firstName, String lastName)
{
    setFirst(firstName);
    setLast(lastName);
    return this;
} // end setName
```

此处，`this` 表示接收对象，它的名字和姓氏刚刚被设置过。

我们可以像调用 `setName` 的 `void` 版本一样调用它的这个定义，或是如下这样调用：

```
Name jill = new Name();
Name myFriend = jill.setName("Jill", "Greene");
```

与之前一样，`setName` 设置了接收对象 `Jill` 的名字和姓氏。然后它返回指向接收对象的引用。这里我们使用一条赋值语句将这个引用保留为 `Jill` 的别名。不过，`setName` 的调用可以作为另一个方法的实参出现。

返回类的一个实例的方法，在 Java 类库的类中并不少见。

静态域和方法

B.27 静态域。有时你需要一个不属于任何一个对象的数据域。例如，一个类可能要记录类中的方法被类的所有对象调用了多少次。这样的数据域称为静态域（static field）、静态变量（static variable）或类变量（class variable）。增加保留字 `static` 就可以声明一个静态域。例如，声明

```
private static int numberofInvocations = 0;
```

定义了类的每个对象都可以访问的 `numberofInvocations` 的一个拷贝。对象可以使用静态域进行相互的通信，或执行某些联合动作。本例中，每个方法都增加 `numberofInvoca-`

tions 的值。这样的静态域通常应该是私有的，以确保只通过适当的访问方法和赋值方法来访问它。

静态域还可用来定义命名常量。语句

```
public static final double YARDS_PER_METER = 1.0936;
```

定义了一个静态域 YARDS_PER_METER。类有 YARDS_PER_METER 的一个拷贝，而不是类的每个对象都有它自己的拷贝，如图 B-8 所示。因为 YARDS_PER_METER 也声明为终极的，它的值不能被改变，所以我们可以安全地让它是公有的。如果你忽略修饰符 final，则静态域一般来说是可以被修改的。

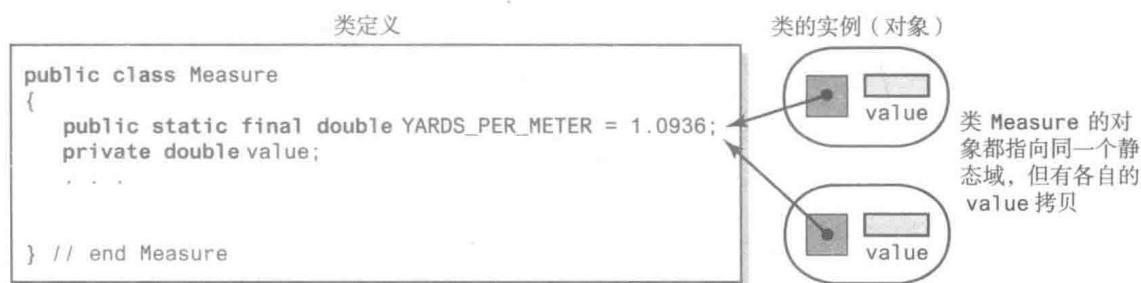


图 B-8 静态域 YARDS_PER_METER 与非静态域 value

注：静态不意味着不变

静态域是被类的所有对象共享的，但它的值可以修改。如果你想让一个域的值保持不变，则必须将它声明为终极的。虽然它们常常一起出现，但修饰符 static 和 final 不是相关的。所以一个域可以是静态的、终极的，或既是静态的又是终极的。

静态方法。有时你需要一个不属于任何一类对象的方法。例如，可能需要一个计算两个整数的最大者的方法，或是一个计算一个数的平方根的方法。这些方法没有明显的应该要属于的对象。这些情形下，可以在方法头添加保留字 static 来定义静态方法。

静态方法 (static method) 或类方法 (class method) 仍是一个类的成员。不过，你使用类名而不是对象名来调用这些方法。例如，Java 预定义的类 Math 中含有几个标准的数学方法，例如 max 和 sqrt。所有这些方法都是静态的，所以你不需要——也真的没什么用——类 Math 的一个对象。使用类名取代对象名来调用这些方法。所以可以写如下的语句

```
int maximum = Math.max(2, 3);
double root = Math.sqrt(4.2);
```

静态方法的定义不能引用类中的任何非静态数据域。但可以引用类中的静态域。同样，它不能调用类的非静态方法，除非它创建类的一个局部对象，并用它来调用非静态方法。但是，静态方法可以调用所在类中的其他静态方法。因为每个应用程序的 main 方法都是静态的，所以这些限制也适用于 main 方法。

程序设计技巧：每个类可以有一个 main 方法

你可以在类定义中包含一个 main 方法用来测试这个类。当你怀疑有错误时，可以很容易地测试这个类定义。因为你——和其他人——可以看到你作了什么测试，你测试中的缺陷变得很明显。如果将类当作一个程序使用，则会调用 main 方法。当你用类来创建另一个类或程序中的对象时，会忽略 main 方法。

**注：构造方法不能是静态的**

构造方法创建类的一个对象，让构造方法是静态的，从而与这样的对象分离是没有意义的。

**学习问题 17 如果不将常数数据域声明为静态的，会怎样？**

重载方法

B.29

同一个类中的几个方法可以有相同的名字，只要方法有不同的形参。因为这些方法的形参在个数或数据类型上不同，所以 Java 能区分它们。我们将这些方法称为**重载的**(overloaded)。

例如，类 Name 有 setName 方法，其方法头是

```
public void setName(String firstName, String lastName)
```

想象一下，我们需要另一个方法，它赋给 Name 对象与另一个 Name 对象相同的名字和姓氏。比如，这个方法的方法头可能是这样的：

```
public void setName(Name otherName)
```

setName 的两个版本不完全相同，因为它们的形参数个数不同。

然后可以用第三个方法重载 setName，其方法头是

```
public void setName(String firstName, Name otherName)
```

想象一下，这个方法将名字设置为 firstName，将姓氏设置为 otherName 的姓。虽然 setName 的三个版本中的两个都带有两个形参，但形参的数据类型不完全一样。两个方法的第一个形参的数据类型是一样的，但第二个形参的数据类型是不同的。

简单修改形参的名字不能算作重载方法。形参的名字是不相关的。当两个方法有相同名字及相同个数的形参时，至少要有一对形参的数据类型必须不同。另外，仅改变返回类型也是不够的。编译程序不能分辨仅有返回类型不同的两个方法。

最后注意，为一个类定义多个构造方法，实际上是重载它们。所以它们的形参必须在个数上或数据类型上不同。

**注：重载一个方法定义**

类中的方法重载同一类中的另一个方法，当两个方法有名字相同但个数或类型不同的形参时。

**注：方法的签名**

方法的签名包括它的名字和形参的名字、类型及次序。所以重载方法有相同的名字但不同的签名。

**学习问题 18 如果方法重载另一个方法，两个方法能有不同的返回类型吗？**

作为类的枚举

如补充材料 1(在线) 段 S1.54 和段 S1.56 中提到, 当编译程序遇到枚举时它创建一个类。本节扩展这一讨论。虽然你应该考虑在程序中使用枚举, 但这些不是本书陈述的核心内容。

当定义一个枚举时, 创建的类有像 `toString`、`equals`、`ordinal` 和 `valueOf` 这样的方法。例如, 我们为扑克牌定义一个简单的枚举, 如下所示。B.30

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
```

则可以以下列方式使用这些方法:

- `Suit.CLUBS.toString()` 返回字符串 CLUBS, 即 `toString` 返回接收对象的名。
- `System.out.println(Suit.CLUBS)` 隐式调用 `toString`, 所以它显示 CLUBS。
- `s.equals(Suit.DIAMONDS)` 测试 `Suit` 的实例 `s` 是否等于 DIAMONDS。
- `Suit.HEARTS.ordinal()` 返回 2, HEARTS 在枚举中的顺序位置。
- `Suit.valueOf("HEARTS")` 返回 `Suit.HEARTS`。

你可以对任意的枚举定义另外的方法——包括构造方法。通过定义一个私有数据域, 可以将值赋给枚举中的每个对象。增加一个 `get` 方法, 将为客户提供访问这些值的方法。程序清单 B-2 列出了枚举 `Suit` 的新定义, 并展示了这些思想是如何实现的。B.31

程序清单 B-2 枚举 Suit

```
1  /** An enumeration of card suits. */
2  enum Suit
3  {
4      CLUBS("black"), DIAMONDS("red"), HEARTS("red"), SPADES("black");
5
6      private final String color;
7
8      private Suit(String suitColor)
9      {
10         color = suitColor;
11     } // end constructor
12
13      public String getColor()
14      {
15         return color;
16     } // end getColor
17 } // end Suit
```

我们选择字符串作为枚举对象的值, 我们设置 CLUBS 的值, 例如, 这样写 CLUBS: `CLUBS("black")`。这个符号调用我们提供的构造方法, 并设置 CLUBS 的私有数据域 `color` 的值为 `black`。注意, `color` 的值不可改变, 因为它声明为终极的。还观察到, 构造方法是私有的, 所以它不能给客户使用。只在 `Suit` 的定义内才可以调用它。方法 `getColor` 提供对 `color` 值的公有访问。



注: 枚举内的构造方法必须是私有的。

程序清单 B-3 中的类提供了前一段中出现的枚举 `Suit` 的简单演示, 补充材料 1(在线) 中的段 S1.63 中描述了这个枚举。除了在 `Suit` 中定义的方法外, 枚举还有方法 `equals`、`ordinal` 和 `valueOf`, 这些方法在本附录段 B.30 和补充材料 1(在线) 段 S1.56 中有所描述。B.32

程序清单 B-3 演示程序清单 B-2 中给出的枚举 Suit 的类

```

1  /** A demonstration of the enumeration Suit. */
2  public class SuitDemo
3  {
4      enum Suit
5      {
6          . . . <See Listing C-2>
7      } // end Suit
8
9      public static void main(String[] args)
10     {
11         for (Suit nextSuit : Suit.values())
12         {
13             System.out.println(nextSuit + " are " + nextSuit.getColor() +
14                     " and have an ordinal value of " +
15                     nextSuit.ordinal());
16         } // end for
17     } // end main
18 } // end SuitDemo

```

输出

```

CLUBS are black and have an ordinal value of 0
DIAMONDS are red and have an ordinal value of 1
HEARTS are red and have an ordinal value of 2
SPADES are black and have an ordinal value of 3

```



注：枚举可以有如 `public` 或 `private` 这样的访问修饰符。如果你省略访问修饰符，则枚举是私有的。在自己的文件中可以定义一个公有枚举，就好像你可以定义任意其他公有类一样。

B.33



示例。补充材料 1 (在线) 的段 S1.54 中定义了一个枚举，用于字符成绩 A、B、C、D 和 F。这里我们扩展那个定义，让它包含加号和减号，以及与那个分数相对应的绩点值。与前面 `Suit` 的定义一样，我们提供私有数据域和一个私有构造方法，表示并初始化每个成绩的字符串表示及数值。还提供每个数据域的访问方法，并重写了方法 `toString`。

程序清单 B-4 中列出了枚举 `LetterGrade` 的新定义。将它定义为公有的，并保存在文件 `LetterGrade.java` 中。

程序清单 B-4 枚举 LetterGrade

```

1  public enum LetterGrade
2  {
3      A("A", 4.0), A_MINUS("A-", 3.7), B_PLUS("B+", 3.3), B("B", 3.0),
4      B_MINUS("B-", 2.7), C_PLUS("C+", 2.3), C("C", 2.0), C_MINUS("C-", 1.7),
5      D_PLUS("D+", 1.3), D("D", 1.0), F("F", 0.0);
6
7      private final String grade;
8      private final double points;
9
10     private LetterGrade(String letterGrade, double qualityPoints)
11     {
12         grade = letterGrade;
13         points = qualityPoints;
14     } // end constructor

```

```

15     public String getGrade()
16     {
17         return grade;
18     } // end getGrade
19
20     public double getQualityPoints()
21     {
22         return points;
23     } // end getQualityPoints
24
25     public String toString()
26     {
27         return getGrade();
28     } // end toString
29 } // end LetterGrade

```

如果定义

```
LetterGrade myGrade = LetterGrade.B_PLUS;
```

则有

- `myGrade.toString()` 返回字符串 B+。
- `System.out.println(myGrade)` 显示 B+, 因为它隐式调用了 `toString`。
- `myGrade.getGrade()` 返回字符串 B+。
- `myGrade.getQualityPoints()` 返回 3.3。

如果没有用我们自己的定义重写方法 `toString`, 则 `myGrade.toString()` 将返回字符串 B_PLUS。

与段 B.31 中给出的枚举 Suit 一样, LetterGrade 也有方法 `equals`、`ordinal` 和 `valueOf`。



学习问题 19 如果 `myGrade` 是 `LetterGrade` 的实例, 且有值 `LetterGrade.B_PLUS`, 则下列每个表达式返回什么?

- `myGrade.ordinal()`
- `myGrade.equals(LetterGrade.A_MINUS)`
- `LetterGrade.valueOf("A_MINUS")`

学习问题 20 下列语句显示什么?

```
System.out.println(LetterGrade.valueOf("A_MINUS"));
```

包

如果将几个相关的类一起放在一个 Java 包 (package) 中, 则使用它们更方便。要将一个类标识为一个特定包的一部分, 可以在包含那个类的文件最前面写如下的语句:

```
package myStuff;
```

然后, 将所有的文件放到一个目录或文件夹中, 并取与包名相同的名字。

要在程序中使用一个包, 可以在程序的最前面写如下的语句

```
import myStuff.*;
```

星号使得包中的所有公有类都能在程序中使用。不过可以将星号替换为包中你想使用的具体

的类名。你可能已经用到了 Java 提供的包，例如包 `java.util`。

为什么我们只说“公有类”？还有什么其他的类吗？你可以使用访问修饰符来控制对类的访问，正如我们可以控制对数据域或方法的访问一样。公有类——不管它是否在包中——对任何其他的类都是可用的。如果你完全省略了类的访问修饰符，则类只能被同一包中其他类使用。这一类类称为有包访问（package access）的。同样，如果你省略数据域或方法的访问修饰符，则它们在同一包内的任意类定义中，可以按名使用，但不能用在包外。在将协作的类组成一个包封装为一个单元的情况下可以使用包访问。如果控制包目录，就能控制谁可以访问包。

Java 类库

B.35

Java 附带了很多类的集合，可以用在你的程序中。例如段 B.28 提到了类 `Math`，它包含几个标准的数学方法，例如 `sqrt`。这个类集合称为 Java 类库（Java Class Library），有时也称为 Java 应用程序接口（Application Programming Interface，API）。这个库中的类组织为标准包。例如类 `Math` 是包 `java.lang` 的一部分。注意，当使用这个包中的类时，不需要使用 `import` 语句。

你应该熟悉为 Java 类库提供的在线文档。

从其他类创建类

先修章节：附录 B

面向对象程序设计的一个主要优点是，当定义新类时可以使用已有类的能力。即使用你或其他人已写的类来创建新类，而不是什么都要自己写。本附录介绍两种方法来实现。

第一种方法，简单地声明一个已有类的实例作为新类的数据域。实际上，如果你曾经定义过一个类，它有一个字符串类型的数据域，你已经是这样做了。因为你的类是由对象组成的，这个技术称为组成。

第二种方法是使用继承，新的类从已有类继承属性和行为，按照需要扩展或修改它们。这个技术比组成更复杂，为此我们将花更多的时间来讨论。与继承在 Java 中的重要性一样，很多情况下，不应该忽视组成也是一项有效及可取的技术，因为继承可能违背了 ADT 的完整性。

组成和继承都在两个类之间定义了关系。这些关系通常分别称为 has a 关系和 is a 关系。当我们在本附录中讨论它们时你会明白原因的。

组成

附录 B 介绍了用 Name 类来表示一个人的名字。它定义了构造方法、访问方法及赋值方法，都涉及一个人的姓氏及名字。Name 中的数据域是 String 类的实例。当类有一个数据域是另一个类的实例时，它使用的是组成（composition）。由于类 Name 有 String 类的实例作为数据域，所以 Name 和 String 之间的关系称为 has a 关系。C.1

现在使用组成来创建另一个类。考虑学生类，每一位学生都有一个名字及一个识别号。所以，类 Student 含有两个对象作为数据域：类 Name 的一个实例及类 String 的一个实例：

```
private Name fullName;  
private String id;
```

图 C-1 显示了 Student 类型的一个对象及它的数据域。注意到，Name 对象有两个 String 对象作为它的数据域。这些数据域实际上含有的是指向对象的引用而不是对象本身，了解这一点很重要。

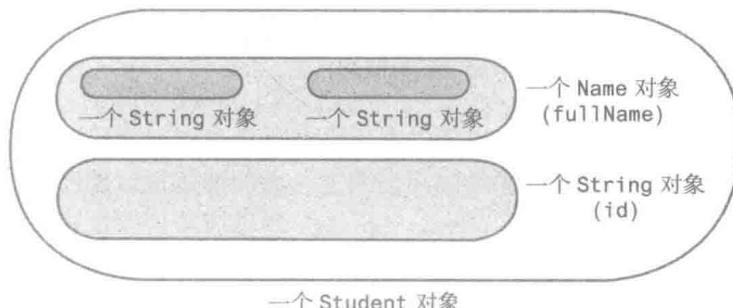


图 C-1 Student 对象是由其他对象组成的

就方法方面，我们让类 `Student` 有构造方法、访问方法、赋值方法及 `toString` 方法。回忆一下，当使用 `System.out.println` 显示对象时将调用 `toString` 方法，所以在你的类定义中要包含一个这样的便利方法。

注：组成 (has a)

当类有对象作为数据域时，它使用的是组成。类的实现中没有专门用于访问这类对象的方法，而且必须像客户那样来操作。即类必须使用对象的方法来操作对象的数据。因为类“has a”或称含有另一个类的一个实例（对象），故称这些类之间有 has a 关系。

C.2 看程序清单 C-1 中 `Student` 类的定义，然后进一步地探讨。

程序清单 C-1 类 `Student`

```

1  public class Student
2  {
3      private Name fullName;
4      private String id;      // Identification number
5
6      public Student()
7      {
8          fullName = new Name();
9          id = "";
10     } // end default constructor
11
12     public Student(Name studentName, String studentId)
13     {
14         fullName = studentName;
15         id = studentId;
16     } // end constructor
17
18     public void setStudent(Name studentName, String studentId)
19     {
20         setName(studentName); // Or fullName = studentName;
21         setId(studentId);    // Or id = studentId;
22     } // end setStudent
23
24     public void setName(Name studentName)
25     {
26         fullName = studentName;
27     } // end setName
28
29     public Name getName()
30     {
31         return fullName;
32     } // end getName
33
34     public void setId(String studentId)
35     {
36         id = studentId;
37     } // end setId
38
39     public String getId()
40     {
41         return id;
42     } // end getId
43
44     public String toString()

```

```

45     {
46         return id + " " + fullName.toString();
47     } // end toString
48 } // end Student

```

当使用默认的构造方法创建一个学生对象时，或是想修改以前赋给学生对象的名字和识别号时，会用到方法 `setStudent`。注意到，这个方法调用了这个类的其他赋值方法来初始化数据域。例如，要将域 `fullName` 设置为参数 `studentName`，`setStudent` 方法使用了下面的语句

```
setName(studentName);
```

还可以将语句写为

```
this.setName(studentName);
```

其中，`this` 指向接收调用方法 `setStudent` 的 `Student` 的实例。或者，也可以写下面这样的赋值语句

```
fullName = studentName;
```

借用于其他方法来实现一个方法通常是可取的。

假定我们想让 `toString` 返回由学生的识别号和姓名组成的一个字符串。它必须使用类 `Name` 中的方法得到字符串形式的姓名。例如，使用下面的语句，`toString` 可以返回所需的字符串

```
return id + " " + fullName.getFirst() + " " + fullName.getLast();
```

或者，写得更简单一些

```
return id + " " + fullName.toString();
```

数据域 `fullName` 指向一个 `Name` 对象，在类 `Student` 的实现中，不能按名访问 `Name` 对象的私有域。我们可以通过访问方法 `getFirst` 和 `getLast`，或是调用 `Name` 的 `toString` 方法来间接访问它们。



学习问题 1 在类 `Address` 的定义中，会用什么数据域来表示学生的地址？

学习问题 2 给类 `Student` 添加一个数据域来表示学生的地址。应该定义什么新方法？

学习问题 3 因为添加了学习问题 2 中所描述的域，必须修改类 `Student` 中的哪些方法？

学习问题 4 附录 B 的段 B.25 中描述的使用 `this` 的默认构造方法的另一种实现是什么？

适配器

假定你有一个类，但它的方法名不适合于你的应用。或者，你想简化某些方法或去掉若干个方法。你可以使用组开来写新的类，它用已有的类的实例作为数据域，并且定义你想要的方法。这样的一个新类称为适配器类 (adapter class)。C.3

例如，假定我们不是使用类 `Name` 的对象来做名字，而是想使用简单的昵称。可以使用字符串来当昵称，但与 `Name` 一样，类 `String` 中含有某些方法是我们不需要的。程序清单 C-2 中的类 `NickName` 使用类 `Name` 的一个实例作为数据域，还有默认的构造方法和 `set` 方法

及 get 方法。不失一般性，我们使用类 Name 的名字域来保存昵称。

程序清单 C-2 类 NickName

```

1  public class NickName
2  {
3      private Name nick;
4
5      public NickName()
6      {
7          nick = new Name();
8      } // end default constructor
9
10     public void setNickName(String nickName)
11     {
12         nick.setFirst(nickName);
13     } // end setNickName
14
15     public String getNickName()
16     {
17         return nick.getFirst();
18     } // end getNickName
19 } // end NickName

```

注意，这个类是如何使用类 Name 的方法来实现它自己的方法的。NickName 对象现在仅有 NickName 的方法，而没有 Name 的方法。



学习问题 5 写语句，定义 bob 为 NickName 的实例，来表示昵称 Bob。然后，使用 bob，写出显示 Bob 的语句。

继承

C.4

继承（Inheritance）是面向对象编程中用来组织类的一个特点。这个名字来自于遗传特征的概念，像是眼睛的颜色、头发的颜色等，但是把继承看作一种分类系统，也许更清楚。继承能让你定义一般的类，之后再添加或修改原来的较一般类定义中的细节，从而定义更具体的类。这能省力气，因为特殊的类继承了一般类的所有特性，你只需对新的或修改后的特性进行编程即可。

例如，你可能为交通工具定义了一个类，然后为一些特定类型的交通工具定义更具体的类，如汽车、马车和船。同样，汽车类又包括轿车类和卡车类。图 C-2 说明了类的这些层次关系。Vehicle 类是如 Automobile 这样的子类（subclasses）的超类（superclass）。Automobile 类是子类 Car 和 Truck 类的超类。超类的另一个术语是基类（base class），子类的另一个术语是派生类（derived class）。

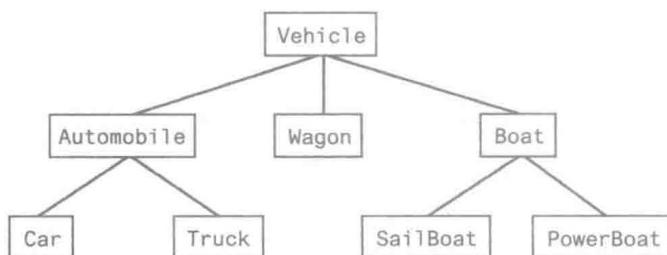


图 C-2 类的层次

在图中向上走时，类更一般化。轿车是汽车，所以它也是交通工具。但是，交通工具不一定是轿车。帆船是船，所以也是交通工具，但交通工具不一定是帆船。

Java 和其他程序设计语言一样，使用继承将类按这种层次结构来组织。程序员可以使用已有的类来写有更多特性的新类。例如，交通工具类具有一定的属性——像是已行驶的英里数——这由其数据域来记录。这个类还有一些行为——像是向前走——这由其方法来定义。类 `Automobile`、`Wagon` 和 `Boat` 也具有这些属性和行为。`Vehicle` 的所有对象都具有的事情，例如向前走的能力，只定义一次，且被 `Automobile`、`Wagon` 和 `Boat` 类来继承。然后子类添加或修改它们所继承的这些属性及行为。没有继承，像是描述向前走这样的行为将不得不在如 `Automobile`、`Wagon`、`Boat`、`Car` 和 `Truck` 等这样的每个子类中重复。

C.5



注：继承

继承是组织类的一种方法，这样，共有的属性和行为仅需为涉及的所有类定义一次。

使用继承，可以定义一般类，之后再添加或修改原来的更一般类中定义的细节从而定义更具体的类。

因为 `Automobile` 类派生于 `Vehicle` 类，所以它继承了那个类的所有数据域及公有方法。`Automobile` 类会有另外的域用来保存像油箱中的油量这样的信息，且它还应该有一些另外的方法。这样的数据域和方法不在 `Vehicle` 类中，因为它们不适用于所有的交通工具。例如，马车没有油箱。

继承机制将超类的所有行为给了子类的实例。例如，汽车能做交通工具能做的每件事情；总之，汽车 is a 交通工具。实际上，继承被称为类间的 is a 关系。因为子类和超类共享属性，所以，仅当子类的实例能够看作超类的实例时使用继承才是有意义的。



注：is a 关系

有了继承，子类的实例也是超类的实例。所以，仅当类之间的 is a 关系有意义时才应该使用继承。



学习问题 6 有些交通工具具有轮子，而有些则没有。修改图 C-2，根据它们是否有轮子来组织交通工具。



示例。我们构造 Java 中继承的示例。假定我们正在设计一个维护学生记录的程序，包括小学、高中和大学。我们可以使用从学生开始的自然层次，来组织不同的学生记录。大学生是学生的一个子类。大学生分为两个更小的子类：本科生和研究生。这些子类可以更进一步地细分为更小的子类。图 C-3 图示了这样的层次安排。

C.6

描述子类的常用办法是使用家族关系的术语。例如，学生类称为本科生类的祖先（ancestor）。反过来。本科生类是学生类的后代（descendant）。

虽然程序中不一定需要一个对应于一般学生的类，但有这样的一个类可能是有用的。例如，所有的学生都有姓名，初始化、修改及显示名字的方法对所有学生也是相同的。在 Java 中，我们可以定义一个类，其中包含属于学生所有子类的属性所用的数据域。这个类同样含有属于所有学生行为对应的方法，包括操作类的数据域的方法。实际上，我们在段 C.2 中已经定义了这样的一个类——`Student`。

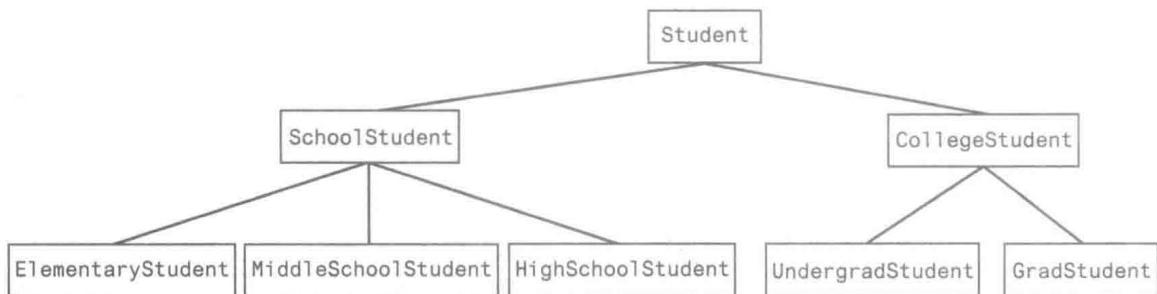


图 C-3 学生类的层次

C.7 现在考虑用于大学生的类。大学生是学生，所以我们使用继承从 `Student` 类来派生 `CollegeStudent` 类。这里，`Student` 是已有的超类，而 `CollegeStudent` 是新子类。子类继承——所以也具有——超类的所有数据域及方法。另外，子类可以定义想要添加的数据域和方法。

要表示 `CollegeStudent` 是 `Student` 的子类，可以在类定义的首行中写短语 `extends Student`。所以 `CollegeStudent` 类定义的开头是这样的

```
public class CollegeStudent extends Student
```

当创建一个子类时，我们仅定义额外的数据域及额外的方法。例如，`CollegeStudent` 类有 `Student` 类的所有数据域和方法，但我们在 `CollegeStudent` 的定义中没有提到这些。具体来说，`CollegeStudent` 类的每个对象都有一个数据域叫 `fullName`，但我们在 `CollegeStudent` 类的定义中没有声明数据域 `fullName`。但是这个数据域是存在的。不过因为 `fullName` 是类 `Student` 的私有数据域，所以我们不能在 `CollegeStudent` 类内直接按名来引用 `fullName`。但是可以使用 `Student` 的方法，访问并修改这个数据域，因为类 `CollegeStudent` 继承了超类 `Student` 中的所有公有方法。

例如，如果 `cs` 是 `CollegeStudent` 的一个实例，则可以写

```
cs.setName(new Name("Joe", "Java"));
```

虽然 `setName` 是超类 `Student` 的方法。因为已经使用继承机制从 `Student` 类来构造 `CollegeStudent`，所以每个本科生 is a 学生。即 `CollegeStudent` 对象“知道”如何执行 `Student` 的行为。

C.8 子类，比如 `CollegeStudent`，还可以在它们从超类继承的部分之外添加一些数据域或方法。例如 `CollegeStudent` 可以添加数据域 `year` 及方法 `setYear` 和 `getYear`。可以设置 `cs` 对象的毕业年份，语句如下

```
cs.setYear(2019);
```

假定还想添加表示所申请学位的数据域及访问和修改它的方法。我们还可以添加用于地址及成绩的数据域，不过为保持简单，我们不加了。现在看看程序清单 C-3 中给出的类，先来看看构造方法。

程序清单 C-3 类 `CollegeStudent`

```

1 public class CollegeStudent extends Student
2 {
3     private int year; // Year of graduation
  
```

```

4     private String degree; // Degree sought
5
6     public CollegeStudent()
7     {
8         super();      // Must be first statement in constructor
9         year = 0;
10        degree = "";
11    } // end default constructor
12
13    public CollegeStudent(Name studentName, String studentId,
14                           int graduationYear, String degreeSought)
15    {
16        super(studentName, studentId); // Must be first
17        year = graduationYear;
18        degree = degreeSought;
19    } // end constructor
20
21    public void setStudent(Name studentName, String studentId,
22                           int graduationYear, String degreeSought)
23    {
24        setName(studentName); // NOT fullName = studentName;
25        setId(studentId);   // NOT id = studentId;
26        // Or setStudent(studentName, studentId); (see Segment C.16)
27
28        year = graduationYear;
29        degree = degreeSought;
30    } // end setStudent
31    <The methods setYear, getYear, setDegree, and getDegree go here.>
32
33    public String toString()
34    {
35        return super.toString() + ", " + degree + ", " + year;
36    } // end toString
37 } // end CollegeStudent

```

在构造方法内调用构造方法

调用超类的构造方法。构造方法通常要初始化类的数据域。在子类中，构造方法如何初始化从超类继承的数据域呢？一种方法是调用超类的构造方法。子类的构造方法可以使用保留字 `super` 当作超类构造方法的名字。

注意到，`CollegeStudent` 类默认构造方法的开头是语句

```
super();
```

这条语句调用超类的默认构造方法。新的默认构造方法必须调用超类的默认构造方法，以对从超类继承的数据域进行正确的初始化。实际上，如果你没有调用 `super`，Java 会为你这样做。本书中，我们总是显式调用 `super`，使程序动作更加清晰。注意，调用 `super` 必须在构造方法的最前面。仅能在其他构造方法内，使用 `super` 来调用构造方法。

以同样的方式，第二个构造方法执行下列语句调用超类中相应的构造方法。

```
super(studentName, studentId);
```

如果省略了这条语句，则 Java 会调用默认构造方法，而这可能不是你想要的。



注：调用超类的构造方法

可以在子类的构造方法定义中，使用 `super` 来显式调用超类的构造方法。这样做时，`super` 必须是构造方法定义中要做的第一个动作。不能使用构造方法的名字来替换

`super`。如果省略了 `super`，则子类的每个构造方法会自动调用超类的默认构造方法。有时，这是你想要的，但有时却不是。



注：构造方法不能继承

类 C 的构造方法创建了其类型为 C 的对象。对这个类来说，有一个名字不是 C 的构造方法是没有意义的。如果类 `CollegeStudent` 继承了 `Student` 的构造方法，会发生的情况是：`CollegeStudent` 类会有一个名为 `Student` 的构造方法。

即使 `CollegeStudent` 类没有继承 `Student` 类的构造方法，它的构造方法也会调用 `Student` 类的构造方法，如你之前所见。

C.10

老调重弹：使用 `this` 来调用构造方法。如你在附录 B 的段 B.25 看到的一样，使用保留字 `this`，就像是这里使用的 `super` 一样，只是它调用同一类的构造方法，而不是调用超类的构造方法。例如，考虑我们在段 C.8 中为 `CollegeStudent` 类添加的构造方法的如下定义：

```
public CollegeStudent(Name studentName, String studentId)
{
    this(studentName, studentId, 0, "");
} // end constructor
```

这个构造方法定义的方法体中的一条语句调用有如下头部的构造方法

```
public CollegeStudent(Name studentName, String studentId,
                      int graduationYear, String degreeSought)
```

与 `super` 一样，使用 `this` 也必须是构造方法定义中的第一个动作。所以，构造方法定义中不能兼有使用 `super` 的调用和使用 `this` 的调用。如果你想既用 `super` 调用又用 `this` 调用，该怎么办呢？这种情况下，你应该使用 `this` 来调用一个以 `super` 作为其第一个动作的构造方法。

超类的私有域和方法

C.11

访问继承的数据域。类 `CollegeStudent` 中有一个 `setStudent` 方法，它带有 4 个形参，`studentName`、`studentId`、`graduationYear` 和 `degreeSought`。要初始化继承的数据域 `fullName` 和 `id`，方法要调用继承的方法 `setName` 和 `setId`：

```
setName(studentName); // NOT fullName = studentName
setId(studentId); // NOT id = studentId
```

回忆 `fullName` 和 `id` 都是定义在超类 `Student` 中的私有数据域。只有类 `Student` 中的方法才能在其定义中按名直接访问 `fullName` 和 `id`。虽然类 `CollegeStudent` 继承了这些数据域，但没有办法能按名访问它们。所以，`setStudent` 方法中不能使用如下这样的赋值语句

```
id = studentId; // ILLEGAL in CollegeStudent's setStudent
```

来初始化数据域 `id`。而是，它必须使用某些公有赋值方法，如 `setId`。



注：超类中私有的数据域不能在任何其他类的方法定义中按名访问，包括子类。虽然这样，子类也还是继承了其超类的数据域。

不能从子类的方法定义内访问超类私有数据域的这一事实，似乎是错误的。但如果不限制

样，访问修饰符 `private` 就变得毫无意义了：任何时候你想访问私有数据域，就能简单地创建一个子类，并在那个类的一个方法中访问就是了。这样，所有的私有数据域都可被任何人花点力气就能访问了。

超类的私有方法。 子类不能直接调用超类的私有方法。这应该不是问题，因为你只应该在私有方法定义所在的类内将它们作为帮助方法来使用。即类的私有方法不是定义行为的。所以，我们说，子类不能继承超类的私有方法。如果你想在子类内使用超类的方法，应该让方法是保护的或是公有的。我们在 Java 插曲 7 中讨论过保护方法。C.12

假定超类 `B` 有一个公有方法 `m`，它调用私有方法 `p`。派生于类 `B` 的类 `D` 继承了公有方法 `m`，但没有继承 `p`。即使这样，当 `D` 的客户调用 `m` 时，`m` 调用了 `p`。所以，超类中的私有方法仍存在，且可用，但是子类不能直接按名调用它。

 **注：**子类不能继承超类的私有方法，也不能按名调用它们。

重写及重载方法

类 `CollegeStudent` 的 `set` 方法和 `get` 方法很简单，故我们不会再花时间去讨论它们。C.13 但是，我们为类提供了方法 `toString`。当新类从超类 `Student` 继承了 `toString` 方法时，我们为什么要这样做？很显然，超类的 `toString` 方法返回的字符串可能含有学生的名字及识别号，但它不含有与子类相关的年份和学位。所以我们需要写一个新的 `toString` 方法。

但是，为什么不让新的方法调用所继承的方法呢？我们可以这样做，但我们需要区分正为 `CollegeStudent` 定义的这个方法和从 `Student` 继承的方法。如你在段 C.8 的类定义中之所见，新的方法 `toString` 含有语句

```
return super.toString() + ", " + degree + ", " + year;
```

因为 `Student` 是超类，所以我们写

```
super.toString()
```

来表示我们正调用超类的 `toString`。如果省略 `super`，则 `toString` 的新版本将调用它自己。这里使用 `super`，就好像它是一个对象一样。相反，使用带括号的 `super` 时，把它看作构造方法定义中的一个方法。

如果返回去看段 C.2，你会看到，`Student` 的 `toString` 方法是下面这样的：

```
public String toString()
{
    return id + " " + fullName.toString();
} // end toString
```

这个方法调用定义在类 `Name` 中的 `toString` 方法，因为对象 `fullName` 是类 `Name` 的实例。

重写一个方法。 在前一段中，我们看到，类 `CollegeStudent` 定义了方法 `toString`，而且还从其超类 `Student` 继承了一个方法 `toString`。这两个方法都没有形参。这样，类中有两个有相同名字、相同参数及相同返回类型的方法。C.14

当子类定义了一个方法，与超类的方法有相同的名字、相同个数及类型的参数及相同的返回类型时，子类中的定义称为**重写 (override)** 了超类中的定义。调用方法的子类对象，将使用子类中的定义。例如，如果 `cs` 是类 `CollegeStudent` 的实例，则

```
cs.toString()
```

使用类 CollegeStudent 中的 `toString` 方法的定义，而不是类 Student 中的 `toString` 的定义，如图 C-4 所示。正如你所见的，子类中的 `toString` 定义可以使用 `super` 来调用超类中的 `toString` 的定义。

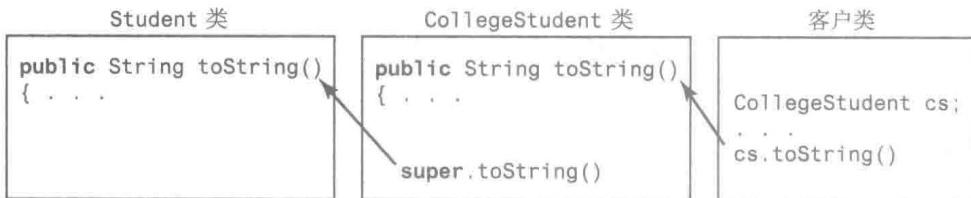


图 C-4 类 CollegeStudent 中的方法 `toString`，重写了 Student 中的 `toString`



注：重写一个方法定义

当子类中的方法与超类中的方法，有相同的名字、相同个数及类型的参数、相同的返回类型时，子类中的方法重写了超类中的方法。因为方法的签名是它的名字和参数，故当两个方法有相同的签名和返回类型时，子类中的方法重写了超类中的方法。



注：重写和访问

子类中的重写方法，可以根据超类中被重写方法的访问权限，而有公有的、保护的或包的访问权限，如下：

| 超类中被重写方法的访问权限 | 子类中重写方法的访问权限 |
|---------------|-----------------------------|
| public | public |
| protected | protected 或 public |
| package | package, protected 或 public |

超类中的私有方法不能被子类的方法重写。

注，Java 插曲 7 中的段 J7.2 中讨论了保护访问，附录 B 中的段 B.34 讨论了包访问。



注：在子类中可以使用 `super` 来调用超类中被重写的方法。



学习问题 7 如果子类重写了超类中的一个保护方法，则重写方法可以是

- a. 公有的？
- b. 保护的？
- c. 私有的？

学习问题 8 如果子类重写了超类中的一个公有方法，则重写方法可以是

- a. 公有的？
- b. 保护的？
- c. 私有的？

学习问题 9 问题 5 要求你创建 `NickName` 的一个实例，来表示昵称 Bob。如果那个对象名为 bob，则下列语句会得到相同的输出吗？请解释之。

```
System.out.println(bob.getNickName());
System.out.println(bob);
```

协变返回类型(可选的)。类不能定义两个有不同返回类型但有相同签名——即相同的名
字和形参——的方法。但是,如果两个方法在不同的类中,且一个类是另一个的子类,则这
是可能的。具体来说,当子类中的方法重写了超类中的方法,它们的签名是相同的。但子类
方法的返回类型,可以是超类方法返回类型的子类。这样的返回类型称为**协变**(covariant)。

例如,在段 C.8 中,类 CollegeStudent 派生于段 C.2 中定义的类 Student。现在假
定类 School 保存 Student 对象的集合。(本书提供能进行相关处理的工具。)类中有方法
getStudent,它根据给定的 ID 号,返回一名学生。这个类的代码可能是这样的:

```
public class School
{
    public Student getStudent(String studentId)
    {
        .
        .
    } // end getStudent
} // end School
```

现在考虑类 College,它表示大学生的集合。可以从 School 派生 College,并重写方
法 getStudent,如下所示:

```
public class College extends School
{
    public CollegeStudent getStudent(String studentId)
    {
        .
        .
    } // end getStudent
} // end College
```

方法 getStudent 与 School 中的 getStudent 有相同的签名,但两个方法的返回
类型不同。事实上,返回类型是协变的——所以是合法的——因为 CollegeStudent 是
Student 的子类。

老调重弹:重载方法。附录 B 中的段 B.29 讨论了同一类中的方法重载。这样的方法有
相同的名字,但签名不同。Java 能区分这些方法,因为它们的形参不同。

假定子类有一个与超类中同名的方法,但方法的参数个数或参数的数据类型不同。子类
将有两个方法——一个是自己定义的,另一个是从超类继承的。子类中的方法重载了超类中
的方法。

例如,超类 Student 和子类 CollegeStudent 都有方法 setStudent。但方法不完全
一样,因为它们的参数个数不同。在 Student 中,方法头是这样的:

```
public void setStudent(Name studentName, String studentId)
```

而在 CollegeStudent 中是这样的:

```
public void setStudent(Name studentName, String studentId,
                      int graduationYear, String degreeSought)
```

类 Student 的实例只可以调用 Student 中的方法,但 CollegeStudent 的实例可以调用两
个类中的方法。再次说明,Java 能够区分两个方法,因为它们有不同的参数。

在 CollegeStudent 类中, setStudent 方法的实现可以使用下列语句,通过调用
Student 的 setStudent 方法,来初始化数据域 fullName 和 id

```
setStudent(studentName, studentId);
```

而不是像我们在段 C.8 中所做的那样调用 `setName` 和 `setId` 方法。因为 `setStudent` 的两个版本有不同的参数列表，所以我们不需要在调用时增加前缀 `super` 来区分两个方法。不过，我们是可以这样写的：

```
super.setStudent(studentName, studentId);
```

注：重载方法定义

类中的一个方法，当它与同类中或超类中的另一个方法，有相同的名字，但形参的个数或类型不同时，就发生了重载。所以重载方法可以有相同的名字，但有不同的签名。

虽然术语“重载”和“重写”很容易混淆，但你应该区分开这两个概念，因为它们同样重要。

C.17

使用多个 `super`。正如我们所说明的，在子类方法的定义中，在方法名前面加上 `super` 和一个点，就可以调用超类中被重写的方法。但如果这个超类本身是从其他超类派生的，你就不能重复使用 `super` 来调用那个超类中的方法。

例如，假定类 `UndergradStudent` 派生于 `CollegeStudent` 类，而后者又派生于 `Student` 类。你或许会认为可以用 `super.super`，在 `Undergraduate` 类的定义中调用 `Student` 类中的方法，比如：

```
super.super.toString(); // ILLEGAL!
```

如注释所注明的，这种重复使用 `super` 在 Java 中是不允许的。

注：super

虽然子类中的方法可以使用 `super` 调用超类中被重写的方法，但方法不能调用在超类的超类中定义的被重写的方法。即 `super.super` 这个写法是不合法的。

 **学习问题 10** 类 `Student` 的两个构造方法（段 C.2）是重载还是重写？为什么？

学习问题 11 如果在类 `CollegeStudent` 中添加方法

```
public void setStudent(Name studentName, String studentId)
```

让它为数据域 `year` 和 `degree` 提供默认值，则你是重载还是重写了类 `Student` 中的 `setStudent`？为什么？

C.18

final 修饰符。假定构造方法调用公有方法 `m`。为简单起见，假定这个方法与构造方法同在类 `C` 中，如下所示：

```
public class C
{
    public C()
    {
        m();
    }
    . . .
} // end default constructor
public void m()
{
    . . .
} // end m
. . .
```

现在假定，从类 C 派生一个新类，并重写方法 m。如果我们调用新类的构造方法，则它会调用超类的构造方法，而后者又会调用方法 m 的重写版本。这个方法可能会用到还没有初始化的数据域，从而引起错误。即使没有发生错误，事实上我们也改变了超类构造方法的行为。

为了规范说明方法定义不能被子类中的新定义重写，可以在方法头中增加 final 修饰符，来定义终极方法 (final method)。例如，可以写

```
public final void m()
```

注意，私有方法自动是终极方法，因为在子类中不能重写它们。

 **程序设计技巧：**如果构造方法调用本类中的公有方法，则将这个方法声明为终极的，这样子类就不能重写这个方法了，也就不能改变构造方法的行为。

构造方法不能是终极的。因为子类不能继承基类中的构造方法，所以也不能重写它，故终极构造方法就没必要了。

可以声明整个类是终极类 (final class)，这种情况下不能将它用作超类，而从它派生任何其他的类。例如，Java 的 String 类就是一个终极类。

 **注：**String 不能是其他任何类的超类，因为它是终极类。

 **程序设计技巧：**当你设计一个类时，考虑现在或未来从它派生的类。它们或许需要访问你类中的数据域。如果你的类没有公有访问方法或赋值方法，则提供这些方法的保护版本。让数据域是私有的。保护访问在 Java 插曲 7 中讨论过。

多重继承

有些程序设计语言允许一个类派生于两个不同的超类。即你可以从类 A 和类 B 派生类 C。被称为多重继承 (multiple inheritance) 的这个特性，在 Java 中是不允许的。在 Java 中，子类只能有一个超类。不过，你可以从类 A 派生类 B，然后再从类 B 派生类 C，因为这不是多重继承。

子类除了能从一个超类派生外，还可以实现任意多个接口——我们在本书的序言中描述过。这个能力使得 Java 具有接近多重继承的能力，但并没有多个超类带来的困难。

类型兼容及超类

子类的对象类型。之前见过 CollegeStudent 类，它从 Student 类派生。现实世界中，每名大学生也是一名学生。这个关系在 Java 中也保持下来。CollegeStudent 类的每个对象也是 Student 类的一个对象。所以，如果有一个方法的参数是 Student 类型的，则调用这个方法的实参可以是 CollegeStudent 类型的一个对象。

具体来说，假定某类中的方法有下面的方法头：

```
public void someMethod(Student scholar)
```

在 someMethod 的方法体中，对象 scholar 可以调用定义在 Student 类中的公有方法。例如，someMethod 的定义中，可以含有表达式 scholar.getId()。也就是说，scholar 有

Student 的行为。

现在考虑 CollegeStudent 的对象 joe。因为 CollegeStudent 类继承了类 Student 的所有公有方法，故 joe 可以调用继承的那些方法。也就是说，joe 的行为可以像 Student 的对象一样。(有时 joe 还可以做其他的，因为它是 CollegeStudent 的对象，但此处并没有这样。) 所以，joe 可以是 someMethod 的实参。即对于对象 o，可以写

```
o.someMethod(joe);
```

这里没有进行自动类型转型^Θ。作为 CollegeStudent 的对象，joe 也是 Student 类型的对象。对象 joe 不需要，且也没有转型为 Student 类的一个对象。

沿着这个思路深入进去。假定从 CollegeStudent 类派生 UndergradStudent 类。现实世界中，每名本科生都是大学生，而每位大学生也是一名学生。这个关系再次在 Java 类中保持下来。UndergradStudent 类的每个对象，也是 CollegeStudent 类的一个对象，所以也是 Student 类的一个对象。故，如果我们有方法，其参数是 Student 类型的，则调用这个方法的实参可以是 UndergradStudent 类的对象。所以，因为继承的原因，一个对象实际上可以有几种类型。

 **注：**子类的对象可以有多种数据类型。适用于祖先类对象的一切，也适用于任何后代类的对象。

C.21 因为子类的对象也有其所有祖先类的类型，所以可以将某个类的对象赋给其任意祖先类的变量，但反过来是不可以的。例如，因为类 UndergradStudent 派生于类 CollegeStudent，而后者又派生于 Student，所以下列语句是合法的：

```
Student amy = new CollegeStudent();
Student brad = new UndergradStudent();
CollegeStudent jess = new UndergradStudent();
```

但是，下列语句是非法的：

```
CollegeStudent cs = new Student();           // ILLEGAL!
UndergradStudent ug = new Student();         // ILLEGAL!
UndergradStudent ug2 = new CollegeStudent(); // ILLEGAL!
```

这非常合情合理。例如，大学生是学生，但学生不一定是大学生。有些程序员发现，短语“is a”在确定对象具有什么类型，及给变量赋什么值是合法的时候非常有用。

 **程序设计技巧：**因为子类的对象也是超类的对象，故当你设计的类与一个已有类之间不存在 is a 关系时没有使用继承。如果你想让类 C 具有类 B 的某些方法，但这些类间不具有 is a 关系，则使用组成来实现。

 **学习问题 12** 如果 HighSchoolStudent 是 Student 的子类，你能将 HighSchoolStudent 的对象赋给 Student 类型的变量吗？为什么？

学习问题 13 你能将 Student 的对象赋给 HighSchoolStudent 类型的变量吗？为什么？

^Θ 补充材料 1 (在线) 中的段 S1.21 回顾了类型转型。

Object 类

正如你已经看到的，如果有一个类 A，且从它派生类 B，然后从类 B 派生类 C，则类 C 的对象可以具有类型 C、类型 B 和类型 A。这对任何子类链都适用，不管链有多长。C.22

Java 有一个类——名为 Object——是每个子类链的链头。这个类是所有其他类的祖先类，甚至是你自己定义的那些类。每个类的每个对象都具有类型 Object，还有它自己类的类型及所有祖先类的类型。如果你的类不是从其他类派生的，则 Java 认为它是从类 Object 派生的。

 注：每个类都是类 Object 类的后代类。

类 Object 含有某些方法，包括 `toString`、`equals` 和 `clone`。每个类都继承了这 3 个方法，或者是直接从 Object 继承，或者是从其他祖先类继承，那些祖先类最终还是从类 Object 继承的。

但继承的方法 `toString`、`equals` 和 `clone`，在你定义的类中几乎从来不能起正确的作用。一般地，你需要使用新的更合适的定义来重写继承的方法。所以当你在类中定义方法时，比如 `toString` 方法，实际上是重写了 Object 类的方法 `toString`。

`toString` 方法。方法 `toString` 不带参数，返回的所有数据应该作为 `String` 的一个对象。不过，你自动得到的数据的字符串表示，不是你想要的格式。继承的 `toString` 方法基于调用对象的内存地址返回一个值。你必须重写 `toString` 的定义，使得它产生一个以合适的格式表示所在类数据的字符串。你或许想再次看看段 C.2 和段 C.8 中的 `toString` 方法。C.23

`equals` 方法。考虑我们在附录 B 中定义的类 `Name` 的下列对象：C.24

```
Name joyce1 = new Name("Joyce", "Jones");
Name joyce2 = new Name("Joyce", "Jones");
Name derek = new Name("Derek", "Dodd");
```

现在 `joyce1` 和 `joyce2` 是两个不同的对象，但含有相同的名字。一般地，我们认为这样的对象应该是相等的，但实际上，`joyce1.equals(joyce2)` 为假。因为 `Name` 类没有定义自己的 `equals` 方法，它使用的是从 `Object` 继承的方法。而 `Object` 类的 `equals` 方法比较对象 `joyce1` 和 `joyce2` 的地址。因为我们有两个不同的对象，故它们的地址是不相等的。不过，`joyce1.equals(joyce1)` 的值为真，因为我们用对象自己来与它进行比较。这个是比较是同一性 (identity) 比较。注意相同与相等是不同的概念。

在 `Object` 类中，方法 `equals` 的定义如下：

```
public boolean equals(Object other)
{
    return (this == other);
} // end equals
```

所以如果 `x` 和 `y` 指向同一个对象，则 `x.equals(y)` 为真。如果想让 `Name` 中的 `equals` 方法有更合适的行为，则必须重写类中的方法。

你能回忆起，`Name` 有两个数据域 `first` 和 `last`，它们都是 `String` 的实例。如果两个 `Name` 对象有相等的姓氏和相等的名字，则我们可以断定这两个对象是相等的。添加到 `Name` 类中的下列方法，通过比较两个 `Name` 对象的数据域来判定它们是否相等：

```
public boolean equals(Object other)
{
```

```

boolean result = false;

if (other instanceof Name)
{
    Name otherName = (Name)other;
    result = first.equals(otherName.first) &&
             last.equals(otherName.last);
} // end if

return result;
} // end equals

```

为确保传给方法 `equals` 的实参是一个 `Name` 对象，可以使用 Java 运算符 `instanceof`。例如，如果 `other` 引用指向 `Name` 类或从 `Name` 类派生的类的实例时，下列表达式

`other instanceof Name`

的值为真。如果 `other` 指向任何其他的类的对象时，或是如果 `other` 是 `null`，则表达式为假。

给定合适的实参，方法 `equals` 比较两个对象的数据域。注意到，我们首先必须将形参 `other` 的类型从 `Object` 转型为 `Name`，以便可以访问 `Name` 的数据域。要比较两个字符串，我们使用 `String` 的 `equals` 方法。类 `String` 定义了自己的 `equals` 方法，它重写了从 `Object` 继承的 `equals` 方法。



学习问题 14 如果 `sue` 和 `susan` 是类 `Name` 的两个实例，什么样的 `if` 语句可以判定它们表示的是不是相同的名字？

C.25

clone 方法。从 `Object` 继承的另一个方法是 `clone` 方法。这个方法不带实参，并返回接收对象的一个拷贝。返回的对象应该与接收对象有相同的数据，但是不同的对象（一个完全相同的双胞胎或一个“克隆”）。与派生于 `Object` 类的其他方法一样，我们必须重写方法 `clone`，这样它才能在我们自己的类中有合适动作。但是，对于 `clone` 方法，我们还要做其他的处理。方法 `clone` 的讨论在 Java 插曲 9 中。

TP312JA
1268(A)

贵大图

2019

小

值

| | | |
|---------------|------|---|
| byte | 1 字节 | -128 ~ 127 |
| short | 2 字节 | -32 768 ~ 32 767 |
| int | 4 字节 | -2 147 483 648 ~ 2 147 483 647 |
| long | 8 字节 | -9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807 |
| 实型 | | |
| float | 4 字节 | $-3.402\ 824 \times 10^{38} \sim 3.402\ 824 \times 10^{38}$ |
| double | 8 字节 | $-1.797\ 693\ 134\ 862\ 32 \times 10^{308} \sim 1.797\ 693\ 134\ 862\ 32 \times 10^{308}$ |
| 字符型 (Unicode) | | |
| char | 2 字节 | 值在 0 ~ 65 535 间的所有 Unicode 字符 |
| 布尔型 | | |
| boolean | 1 位 | true, false |

Unicode 字符编码

显示的可打印字符是 Unicode 字符集的一个子集，称为 ASCII 字符集。无论字符是 Unicode 字符集的成员还是 ASCII 字符集的成员，编号都是一样的（字符编号 32 是空白）。

| | | | | | | | |
|----|----|----|---|-----|---|-----|---|
| 32 | | 56 | 8 | 80 | P | 104 | h |
| 33 | ! | 57 | 9 | 81 | Q | 105 | i |
| 34 | " | 58 | : | 82 | R | 106 | j |
| 35 | # | 59 | ; | 83 | S | 107 | k |
| 36 | \$ | 60 | < | 84 | T | 108 | l |
| 37 | % | 61 | = | 85 | U | 109 | m |
| 38 | & | 62 | > | 86 | V | 110 | n |
| 39 | ' | 63 | ? | 87 | W | 111 | o |
| 40 | (| 64 | @ | 88 | X | 112 | p |
| 41 |) | 65 | A | 89 | Y | 113 | q |
| 42 | * | 66 | B | 90 | Z | 114 | r |
| 43 | + | 67 | C | 91 | [| 115 | s |
| 44 | , | 68 | D | 92 | \ | 116 | t |
| 45 | - | 69 | E | 93 |] | 117 | u |
| 46 | . | 70 | F | 94 | ^ | 118 | v |
| 47 | / | 71 | G | 95 | _ | 119 | w |
| 48 | 0 | 72 | H | 96 | ` | 120 | x |
| 49 | 1 | 73 | I | 97 | a | 121 | y |
| 50 | 2 | 74 | J | 98 | b | 122 | z |
| 51 | 3 | 75 | K | 99 | c | 123 | { |
| 52 | 4 | 76 | L | 100 | d | 124 | |
| 53 | 5 | 77 | M | 101 | e | 125 | } |
| 54 | 6 | 78 | N | 102 | f | 126 | ~ |
| 55 | 7 | 79 | O | 103 | g | | |

数据结构与抽象 Java语言描述 原书第5版

Data Structures and Abstractions with Java Fifth Edition

本书是数据结构的经典教材，内容涵盖线性结构、层次结构、图等数据结构，排序、查找等重要方法，以及算法的评估和迭代/递归实现方式等，并采用Java语言基于数组和链表实现了各种ADT。本书的组织方式独特，语言简洁，示例丰富，程序规范，习题多样。

本书新特色

- 新增加了一章讨论递归，介绍语法、语言及回溯。
- 增加了新的设计决策、注、安全说明及编程技巧。
- 在大部分章节中，增加了侧重游戏、电子商务及财务的练习和程序设计项目。
- 调整了某些主题的次序，相关的主题集中介绍，内容连贯。
- 修改了插图，使之更容易阅读和理解。
- 将“自测题”改名为“学习问题”，并将答案移至网站中，鼓励小组一起讨论答案。
- 书中包含了关于Java类的附录，而不是将其放在网站中。

作者简介

弗兰克·M. 卡拉诺 (Frank M. Carrano) 是美国罗得岛大学计算机科学系荣誉退休教授，1969年获得美国锡拉丘兹大学计算机科学专业博士学位。他的研究兴趣包括数据结构、计算机科学教育、社会问题的计算处理和数值计算。Carrano教授撰写了多本著名的计算机科学高年级本科生教科书。

蒂莫西·M. 亨利 (Timothy M. Henry) 是美国新英格兰理工学院计算机科学系副教授，1986年获得美国欧道明大学计算机科学专业硕士学位，2001年获得美国罗得岛大学应用数学专业博士学位，自2000年以来一直保有美国PMI的项目管理专家 (Project Management Professional, PMP) 认证资格。他教授的课程有数据结构与抽象、编程语言基础、操作系统与网络、计算机系统基础、计算机科学项目、文件系统取证等，研究领域涉及计算机和数学取证、交互式3D图形关系、传感器网络。



P Pearson

www.pearson.com

华章教育服务微信号

投稿热线: (010) 88379604

读者信箱: hzjsj@hzbook.com

客服电话: (010) 88361066 88379833 68326294

华章网站: www.hzbook.com

网上购书: www.china-pub.com

数字阅读: www.hzmedia.com.cn

上架指导: 计算机/数据结构

ISBN 978-7-111-63637-3



9 787111 636373 >

定价: 139.00元